International
Virtual
Observatory
Alliance

SAMP — Simple Application Messaging Protocol
Version 1.3

IVOA Recommendation 2012-04-11

This version:

1.3: Recommendation 2012-04-11 (Revision 1684)

Latest version:

`http://www.ivoa.net/Documents/latest/SAMP.html`

Previous versions:

Working Group:

Applications

Editor(s):

T. Boch, M. Fitzpatrick, M. Taylor

Authors:

M. Taylor (m.b.taylor@bristol.ac.uk)
T. Boch (boch@astro.u-strasbg.fr)
M. Fitzpatrick (fitz@noao.edu)
A. Allan (aa@astro.ex.ac.uk)
J. Fay (jfay@microsoft.com)
L. Paioro (luigi@lambrate.inaf.it)
J. Taylor (jontayler@gmail.com)
D. Tody (dtody@nrao.edu)

Abstract

SAMP is a messaging protocol that enables astronomy software tools to interoperate and communicate.

IVOA members have recognised that building a monolithic tool that attempts to fulfil all the requirements of all users is impractical, and it is a better use of our limited resources to enable individual tools to work together better. One element of this is defining common file formats for the exchange of data between different applications. Another important component is a messaging system that enables the applications to share data and take advantage of each other's functionality. SAMP supports communication between applications on the desktop and in web browsers, and is also intended to form a framework for more general messaging requirements.

Status of this Document

This document has been produced by the IVOA Applications Working Group. It has been reviewed by IVOA Members and other interested parties, and has been endorsed by the IVOA Executive Committee as an IVOA Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. IVOA's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability inside the Astronomical Community.

Comments, questions and discussions relating to this document may be posted to the mailing list of the SAMP subgroup of the Applications Working Group, apps-samp@ivoa.net. Supporting material and further discussion may be found at `http://www.ivoa.net/samp/`.

Changes since earlier versions may be found in Appendix B.

Contents

1 Introduction

1.1 Non-Technical Preamble and Position in IVOA Architecture

SAMP, the Simple Application Messaging Protocol, is a standard for allowing software tools to exchange control and data information, thus facilitating tool interoperability, and so allowing users to treat separately developed applications as an integrated suite. An example of an operation that SAMP might facilitate is passing a source catalogue from one GUI application to another, and subsequently allowing sources marked by the user in one of those applications to be visible as such in the other.

The protocol has been designed, and implementations developed, within the context of the International Virtual Observatory Alliance (IVOA), but

the design is not specific either to the Virtual Observatory (VO) or to Astronomy. It is used in practice for both VO and non-VO work with astronomical tools, and is in principle suitable for non-astronomical purposes as well.

The SAMP standard itself is neither a dependent, nor a dependency, of other VO standards, but it provides valuable glue between user-level applications which perform different VO-related tasks, and hence contributes to the integration of Virtual Observatory functionality from a user's point of view. Figure 1 illustrates SAMP in the context of the IVOA Architecture [1]. Most existing tools which operate in the User Layer of this architecture provide SAMP interoperability.
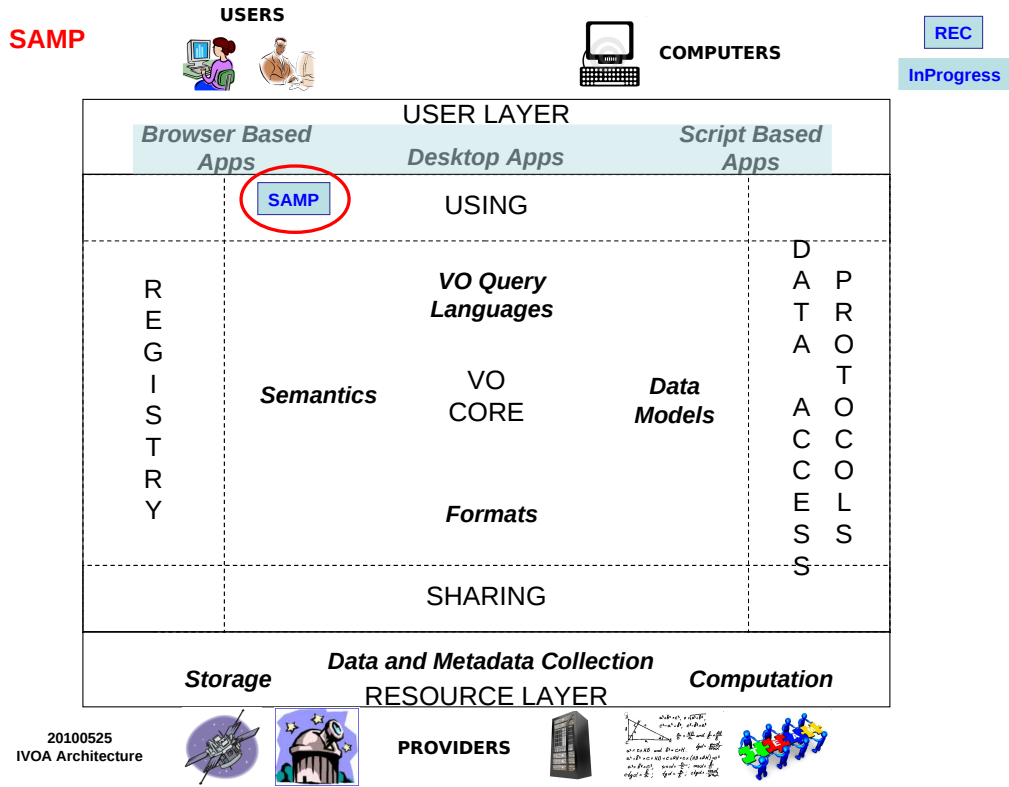
Figure 1: IVOA Architecture diagram [1]. The SAMP protocol appears in the "Using" region.

The semantics of messages that can be exchanged using SAMP are defined by contracts known as MTypes (message-types), which are defined by developer agreement outside of this standard. The list of MTypes used for common astronomical and VO purposes can be found near `http://www.ivoa.net/samp/`; many of these make use of standards from elsewhere in

the IVOA Architecture, including VOTable, VOResource, Simple Spectral Access, UCD and Utype.

1.2 History

SAMP, the Simple Application Messaging Protocol, is a direct descendent of the PLASTIC protocol, which in turn grew — in the European VOTech framework — from the interoperability work of the Aladin [2] and VisIVO [3] teams. We also note the contribution of the team behind the earlier XPA protocol [4]. For more information on PLASTIC's history and purpose see the IVOA Note *PLASTIC — a protocol for desktop application interoperability* [5] and the PLASTIC SourceForge site [6].

SAMP has similar aims to PLASTIC, but incorporates lessons learnt from two years of practical experience and ideas from partners who were not involved in PLASTIC's initial design.

Broadly speaking, SAMP is an abstract framework for loosely-coupled, asynchronous, RPC-like and/or event-based communication, based on a central service providing multi-directional publish/subscribe message brokering. The message semantics are extensible and use structured but weakly-typed data. These concepts are expanded on below. It attempts to make as few assumptions as possible about the transport layer or programming language with which it is used. It also defines a "Standard Profile" which specifies how to implement this framework using XML-RPC [7] as the transport layer. The result of combining this Standard Profile with the rest of the SAMP standard is deliberately similar in design to PLASTIC, and this has been largely successful in its intention of enabling PLASTIC applications to be modified to use SAMP instead without great effort. More recently (version 1.3) an additional "Web Profile" has been introduced, in order to facilitate use of SAMP from web applications.

1.3 Requirements and Scope

SAMP aims to be a simple and extensible protocol that is platform- and language-neutral. The emphasis is on a simple protocol with a very shallow learning curve in order to encourage as many application authors as possible to adopt it. SAMP is intended to do what you need most of the time. The SAMP authors believe that this is the best way to foster innovation and collaboration in astronomy applications.

It is important to note therefore that SAMP's scope is reasonably modest; it is not intended to be the perfect messaging solution for all situations.

In particular SAMP itself has no support for transactions, security, or guaranteed message delivery or integrity. However, by layering the SAMP architecture on top of suitable messaging infrastructures such capabilities could be provided. These possibilities are not discussed further in this document, but the intention is to provide an architecture which is sufficiently open to allow for such things in the future with little change to the basics.

1.4 Types of Messaging

SAMP is currently targetted at inter-application desktop messaging with the idea that the basic framework presented here is extensible to meet future needs, and so it is beyond the scope of this document to outline the many types of messaging systems in use today (these are covered in detail in many other documents). While based on established messaging models, SAMP is in many ways a hybrid of several basic messaging concepts; the protocol is however flexible enough that later versions should be able to interact fairly easily with other messaging systems because of the shared messaging models.

The messaging concepts used within SAMP include:

Publish/Subscribe Messaging: A publish/subscribe (pub/sub) messaging system supports an event driven model where information producers and consumers participate in message passing. SAMP applications "publish" a message, while consumer applications "subscribe" to messages of interest and consume events. The underlying messaging system routes messages from producers to consumers based on the message types in which an application has registered an interest.

Point-to-Point Messaging: In point to point messaging systems, messages are routed to an individual consumer which maintains a queue of "incoming" messages. In a traditional message queue, applications send messages to a specified queue and clients retrieve them. In SAMP, the message system manages the delivery and routing of messages, but also permits the concept of a directed message meant for delivery to a specific application. SAMP does not, however, guarantee the order of message delivery as with a traditional message queue.

Event-based Messaging: Event-based systems are systems in which producers generate events, and in which messaging middleware delivers events to consumers based upon a previously specified interest. One typical usage pattern of these systems is the publish/subscribe paradigm, however these systems are also widely used for integrating loosely coupled application components. SAMP allows for the concept that an "event" occurred in the system and that these message types may have

requirements different from messages where the sender is trying to invoke some action in the network of applications.

Synchronous vs. Asynchronous Messaging: As the term is used in this document, a "synchronous" message is one which blocks the sending application from further processing until a reply is received. However, SAMP messaging is based on "asynchronous" message and response in that the delivery of a message and its subsequent response are handled as separate activities by the underlying system. With the exception of the synchronous message pattern supported by the system, sending or replying to a message using SAMP allows an application to return to other processing while the details of the delivery are handled separately.

1.5   About this Document

This document contains the following main sections describing the SAMP protocol and how to use it. Section 2 covers the requirements, basic concepts and overall architecture of SAMP. Section 3 defines abstract (i.e. independent of language, platform and transport protocol) interfaces which clients and hubs must offer to participate in SAMP messaging, along with data types and encoding rules required to use them. Section 4 explains how the abstract API can be mapped to specific network operations to form an interoperable messaging system, and defines the "Standard Profile", based on XML-RPC, which gives a particular set of such mappings suitable for general purpose desktop applications. Section 5 defines the "Web Profile", an alternative mapping suitable for web applications. Section 6 describes the use of the MType keys used to denote message semantics, and outlines an MType vocabulary.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [8].

2   Architectural Overview

This section provides a high level view of the SAMP protocol.

2.1   Nomenclature

In the text that follows these terms are used:

Hub: A broker service for routing SAMP Messages.

Client: An application that talks to a Hub using SAMP. May be a Sender, Recipient, or both.

Sender: A Client that sends a SAMP Message to one or more Recipients via the Hub.

Recipient: A Client that receives a SAMP Message from the Hub. This may have originated from another Client or from the Hub itself.

Message: A communication sent from a Sender to a Recipient via a SAMP Hub. Contains an MType and zero or more named parameters. May or may not provoke a Response.

Response: A communication which may be returned from a Recipient to a Sender in reply to a previous Message. A Response may contain returned values and/or error information. In the terminology of this document, a Response is not itself a Message. A Response is also known as a Reply in this document.

MType: A string defining the semantics of a Message and of its arguments and return values (if any). Every Message contains exactly one MType, and a Message is only delivered to Clients subscribed to that MType.

Subscription: A Client is said to be Subscribed to a given MType if it has declared to the Hub that it is prepared to receive Messages with that MType.

Callable Client: A Client to which the Hub is capable of performing callbacks. Clients are not obliged to be Callable, but only Callable Clients are able to receive Messages or asynchronous Responses.

Broadcast: To send a SAMP Message to all Subscribed Clients excluding the Sender.

Profile: A set of rules which map the abstract API defined by SAMP to a set of I/O operations which may be used by Clients to send and receive actual Messages.

2.2 Messaging Topology

SAMP has a hub-based architecture (see Figure 2). The hub is a single service used to route all messages between clients. This makes application discovery more straightforward in that each client only needs to locate the hub, and the services provided by the hub are intended to simplify the actions of the client. A disadvantage of this architecture is that the hub may be a message bottleneck and potential single point of failure. The former means that SAMP may not be suitable for extremely high throughput requirements; the latter may be mitigated by an appropriate strategy for hub restart if failure is likely.

Note that the hub is defined as a service interface which may have any of
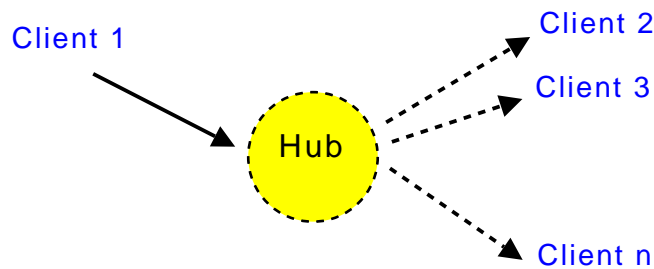
Figure 2: The SAMP hub architecture

a number of implementations. It may be an independent application running as a daemon, an adapter interface layered on top of an existing messaging infrastructure, or a service provided by an application which is itself one of the hub's clients.

2.3 The Lifecycle of a Client

A SAMP client goes through the following phases:

1. Determine whether a hub is running by using the appropriate hub discovery mechanism.
2. If so, use the hub discovery mechanism to work out how to communicate with the hub.
3. Register with the hub.
4. Store metadata such as client name, description and icon in the hub.
5. Subscribe to a list of MTypes to define messages which may be received.
6. Interrogate the hub for metadata of other clients.
7. Send and/or receive messages to/from other clients via the hub.
8. Unregister with the hub.

Phases 4–7 are all optional and may be repeated in any order.

By subscribing to the MTypes described in Section 6.4.1 a client may, if it wishes, keep track of the details of other clients' registrations, metadata and subscriptions.

2.4 The Lifecycle of a Hub

A SAMP hub goes through the following phases:

1. Locate any existing hub by using the appropriate hub discovery mechanism.

    (a) Check whether the existing hub is alive.

    (b) If so, exit.

2. If no hub is running, or a hub is found but is not functioning, write/overwrite the hub discovery record and start up.

3. Await client registrations. When a client makes a legal registration, assign it a public ID, and add the client to the table of registered clients under the public ID. Broadcast a message announcing the registration of a new client.

4. When a client stores metadata in the hub, broadcast a message announcing the change and make the metadata available.

5. When a client updates its list of subscribed MTypes, broadcast a message announcing the change and make the subscription information available

6. When the hub receives a message for relaying, pass it on to appropriate recipients which are subscribed to the message's MType. Broadcast messages are sent to all subscribed clients except the sender, messages with a specified recipient are sent to that recipient if it is subscribed.

7. Await client unregistrations. When a client unregisters, broadcast a message announcing the unregistration and remove the client from the table of registered clients.

8. If the hub is unable to communicate with a client, it may unregister it as described in phase 7.

9. When the hub is about to shutdown, broadcast a message to all subscribed clients.

10. Delete the hub discovery record.

Phases 3–8 are responses to events which may occur multiple times and in any order.

The MTypes broadcast by the hub to inform clients of changes in its state are given in Section 6.4.1.

Readers should note that, given this scheme, race conditions may occur. A client might for instance try to register with a hub which has just shut down, or attempt to send to a recipient which has already unregistered. Specific profiles MAY define best-practice rules in order to best manage these conditions, but in general clients should be aware that SAMP's lack of guaranteed message delivery and timing means that unexpected conditions are possible.

2.5 Message Delivery Patterns

Messages can be sent according to three patterns, differing in whether and how a response is returned to the sender:

1. Notification
2. Asynchronous Call/Response
3. Synchronous Call/Response

The Notification pattern is strictly one-way while in the Call/Response patterns the recipient returns a response to the sender.

If the sender expects to receive some useful data as a result of the receiver's processing, or if it wishes to find out whether and when the processing is completed, it should use one of the Call/Response variants. If on the other hand the sender has no interest in what the recipient does with the message once it has been sent, it may use the Notification pattern. Notification, since it involves no communication back from the recipient to the sender, uses fewer resources. Although typically "event"-type messages will be sent using Notify and "request-for-information"-type messages will be sent using Call/Response, the choice of which delivery pattern to use is entirely distinct from the content of the message, and is up to the sender; any message (MType) may be sent using any of the above patterns. Apart from the fact of returning or not returning a response, the recipient SHOULD process messages in exactly the same way regardless of which pattern is used.

From the receiver's point of view there are only two cases: Notification and Asynchronous Call/Response. However, the hub provides a convenience method which simulates a synchronous call from the sender's point of view. The purpose of this is to simplify the use of the protocol in situations such as scripting environments which cannot easily handle callbacks. However, it is RECOMMENDED to use the asynchronous pattern where possible due to its greater robustness.

2.6 Extensible Vocabularies

At several places in this document structured information is conveyed by use of a controlled but extensible vocabulary. Some examples are the client metadata keys (Section 3.6), message encoding keys (Section 3.8) and MType names (Section 6).

Wherever this pattern is used, the following rules apply. This document defines certain well-known keys with defined meanings. These may be OPTIONAL or REQUIRED as documented, but if present MUST be used by

clients and hubs in the way defined here. All such well-known keys start with the string "`samp.`".

Clients and hubs are however free to introduce and use non-well-known keys as they see fit. Any string may be used for such a non-standard key, with the restriction that it MUST NOT start with the prefix "`samp.`". The prefix "`x-samp.`" has a special meaning as described below.

The general rule is that hubs and clients encountering keys which they do not understand SHOULD ignore them, propagating them to downstream consumers if appropriate. As far as possible, where new keys are introduced they SHOULD be such that applications ignoring them will continue to behave in a sensible way.

Hubs and clients are therefore able to communicate information additional to that defined in the current version of this document without disruption to those which do not understand it. This extensibility may be of use to applications which have mutual private requirements outside the scope of this specification, or to enable experimentation with new features. If the SAMP community finds such experiments useful, future versions of this document may bring such functionality within the SAMP specification itself by defining new keys in the "`samp.`" namespace. The ways in which these vocabularies are used means that such extensions should be possible with minimal upheaval to the existing specification and implementations.

Non-well-known keys (those outside of the "`samp`" namespace) fall into two categories: those which are candidates for future incorporation into the SAMP standard as well-known, and those which are not. If developers are experimenting with keys which they hope or believe may be incorporated into the SAMP standard as well-known at some time in the future, they may use the special namespace "`x-samp`". If a future version of the standard does incorporate such a key as well-known, the prefix is simply changed from "`x-samp.`" to "`samp.`". Consumers of such keys SHOULD treat keys which differ only in the substitution of the prefix "`samp.`" for "`x-samp.`" or vice versa as if they have identical semantics, so for instance a client application should treat the value of a metadata item with key "`x-samp.a.b`" in exactly the same way as one with key "`samp.a.b`". The "`samp`" and "`x-samp`" form of the same key SHOULD NOT be presented in the same map. If both are presented together, the "`samp`" form MAY be considered to take precedence, though any reasonable behaviour is permitted. This scheme makes it easy to introduce new well-known keys in a way which neither makes illicit use of the reserved "`samp.`" namespace nor requires frequent updates to the SAMP standard, and which places a minimum burden on application developers. Lists of keys in the "`x-samp`" namespace under discussion may be found near `http://www.ivoa.net/samp/`.

2.7 Use of Profiles

The design of SAMP is based on the abstract interfaces defined in Section 3. On its own however, this does not include the detailed instructions required by application developers to achieve interoperability. To achieve that, application developers must know how to map the operations in the abstract SAMP interfaces to specific I/O (in most cases, network) operations. It is these I/O operations which actually form the communication between applications. The rules defining this mapping from interface to I/O operations are what constitute a SAMP "Profile" (the term "Implementation" was considered for this purpose, but rejected because it has too many overlapping meanings in this context).

There are two ways in which such a Profile can be specified as far as client application developers are concerned:

1. By describing exactly what bytes are to be sent using what wire protocols for each SAMP interface operation
2. By providing one or more language-specific libraries with calls which correspond to those of the SAMP interface

Although either is possible, SAMP is well-suited for approach (1) above given a suitable low-level transport library. This is the case since the operations are quite low-level, so client applications can easily perform them without requiring an independently developed SAMP library. This has the additional advantages that central effort does not have to be expended in producing language-specific libraries, and that the question of "unsupported" languages does not arise.

Splitting the abstract interface and Profile descriptions in this way separates the basic design principles from the details of how to apply them, and it opens the door for other Profiles serving other use cases in the future.

This document defines two profiles along the lines of (1) above. The Standard Profile (Section 4) which dates from the first version of this document, is suitable for desktop applications, while the Web Profile (Section 5), introduced at SAMP version 1.3, is suitable for web (browser-based) applications.

A client author will usually only need to implement SAMP communications using a single profile. Hub implementations should ideally implement all known profiles; in this way clients using different profiles can communicate transparently with each other via a hub which mediates between them. Since the different profiles are based on the same abstract interface (Section 3), such mediation will not lead to loss or distortion of the communications.

2.8 Security Considerations

SAMP enables inter-process communications including the capability for one client to cause execution of code by another client. This raises the possibility of an unprivileged client performing privileged actions in virtue of its SAMP-enabled interoperation. Whether this is problematic in practice depends on two things: first the identities of the interoperating clients (whether they all share similar levels of privilege or trust) and second the semantics of the messages (the nature of the code that may be executed remotely, and particularly how it can be parameterised). In the case that untrusted clients can cause execution of potentially damaging code by trusted clients, there is a serious security issue.

The trustedness of registered clients is determined by the profile or profiles operated by the hub at a given time (Section 2.7), since the extent to which registered clients are trusted may differ between different profiles. Clients registering via the Standard Profile in its usual configuration can be assumed all to be owned by the same user and hence to have the same privileges (Section 4.3.2), but Web Profile clients usually have only limited access privileges outside of the interoperability granted by SAMP (Section 5.4).

In most cases profiles will, in virtue of their definition or at least of their implementation, provide reasonable assurance that registered clients are unlikely to be hostile. However for clients which may be run in a general SAMP context, it is wise not to expose via SAMP mechanisms unrestricted access to sensitive resources. In particular, it is recommended not to introduce MTypes which can be made to execute arbitrary code (inviting injection attacks), or to declare metadata which reveals sensitive information. As an alternative approach, it may be appropriate in certain usage scenarios to ensure that only a restricted secure profile is running.

3 Abstract APIs and Data Types

3.1 Hub Discovery Mechanism

In order to keep track of which hub is running, a hub discovery mechanism, capable of yielding information about how to determine the existence of and communicate with a running hub, is needed. This is a Profile-specific matter and specific prescriptions are described in Sections 4.3 (Standard Profile) and 5.2.3 (Web Profile).

3.2 Communicating with the Hub

The details of how a client communicates with the hub are Profile-specific. Specific prescriptions are described in Sections 4 (Standard Profile) and 5 (Web Profile).

3.3 SAMP Data Types

For all hub/client communication, including the actual content of messages, SAMP uses three conceptual data types:

1. `string` — a scalar value consisting of a sequence of characters; each character is an ASCII character with hex code 09, 0a, 0d or 20–7f
2. `list` — an ordered array of data items
3. `map` — an unordered associative array of key-value pairs, in which each key is a `string` and each value is a data item

These types can in principle be nested to any level, so that the elements of a list or the values of a map may themselves be strings, lists or maps.

There is no reserved representation for a null value, and it is illegal to send a null value in a SAMP context even if the underlying transport protocol permits this. However a zero-length string or an empty list or map may, where appropriate, be used to indicate an empty value.

Although SAMP imposes no maximum on the length of a string, particular transport protocols or implementation considerations may effectively do so; in general, hub and client implementations are not expected to deal with data items of unlimited size. General purpose MTypes SHOULD therefore be specified so that bulk data is not sent within the message or response. In general it is preferred to define a message parameter or result element as the URL or filename of a potentially large file rather than as the inline text of the file itself. SAMP defines no formal list of which URL protocols are permitted in such cases, but clients which need to dereference such URLs SHOULD be capable of dealing with at least the "http" and "file" schemes. "https", "ftp" and other schemes are also permitted, but when sending such URLs, consideration should be given to whether receiving clients are likely to be able to dereference them.

At the protocol level there is no provision for typing of scalars. Unlike many Remote Procedure Call (RPC) protocols SAMP does not distinguish syntactically between strings, integers, floating point values, booleans etc. This minimizes the restrictions on what underlying transport protocols may be used, and avoids a number of problems associated with using typed values from weakly-typed languages such as Python and Perl. The practical

requirement to transmit these types is addressed however by the next section.

3.4   Scalar Type Encoding Conventions

Although the protocol itself defines `string` as the only scalar type, some MTypes will wish to define parameters or return values which have non-string semantics, so conventions for encoding these as `string`s are in practice required. Such conventions only need to be understood by the sender and recipient of a given message and so can be established on a per-MType basis, but to avoid unnecessary duplication of effort this section defines some commonly-used type encoding conventions.

We define the following BNF productions:

```
<digit>        ::= "0" | "1" | "2" | "3" | "4" | "5" | "6"
                 | "7" | "8" | "9"
<digits>       ::= <digit> | <digits> <digit>
<float-digits> ::= <digits> | <digits> "." | "." <digits>
                 | <digits> "." <digits>
<sign>         ::= "+" | "-"
```

With reference to the above we define the following type encoding conventions:

- `<SAMP int> ::= [ <sign> ] <digits>`
  An integer value is encoded using its decimal representation with an OPTIONAL preceding sign and with no leading, trailing or embedded whitespace. There is no guarantee about the largest or smallest values which can be represented, since this will depend on the processing environment at decode time.

- `<SAMP float> ::= [ <sign> ] <float-digits>`
  `                 [ "e" | "E" [ <sign> ] <digits> ]`
  A floating point value is encoded as a mantissa with an OPTIONAL preceding sign followed by an OPTIONAL exponent part introduced with the character "`e`" or "`E`". There is no guarantee about the largest or smallest values which can be represented or about the number of digits of precision which are significant, since these will depend on the processing environment at decode time.

- `<SAMP boolean> ::= "0" | "1"`
  A boolean value is represented as an integer: zero represents false, and

17

any other value represents true. 1 is the RECOMMENDED value to represent true.

The numeric types are based on the syntax of the C programming language, since this syntax forms the basis for typed data syntax in many other languages. There may be extensions to this list in future versions of this standard.

Particular MType definitions may use these conventions or devise their own as required. Where the conventions in this list are used, message documentation SHOULD make it clear using a form of words along the lines "this parameter contains a *SAMP int*".

3.5   Registering with the Hub

A client registers with the hub to:

1. establish communication with the hub
2. advertise its presence to the hub and to other clients
3. obtain registration information

The registration information is in the form of a `map` containing data items which the client may wish to use during the SAMP session. The hub MUST fill in values for the following keys in the returned `map`:

`samp.hub-id` — The client ID which is used by the hub when it sends messages itself (rather than forwarding them from other senders). For instance, this ID will be used when the hub sends the `samp.hub.event.shutdown` message.

`samp.self-id` — The client ID which identifies the registering client.

These keys form part of an extensible vocabulary as explained in Section 2.6. In most cases a client will not require either of the above IDs for normal SAMP operation, but they are there for clients which do wish to know them. Particular Profiles may require additional entries in this map.

Immediately following registration, the client will typically perform some or all of the following OPTIONAL operations:

- supply the hub with metadata about itself, using the `declareMetadata()` call
- tell the hub how it wishes the hub to communicate with it, if at all (the mechanism for this is profile-dependent, and it may be implicit in registration)
- inform the hub which MTypes it wishes to subscribe to, using the `declareSubscriptions()` call

18

3.6 Application Metadata

A client may store metadata in the form of a `map` of key-value pairs in the hub for retrieval by other clients. Typical metadata might be the human-readable name of the application, a description and a URL for its icon, but other values are permitted. The following keys are defined for well-known metadata items:

`samp.name` — A one word title for the application.
`samp.description.text` — A short description of the application, in plain text.
`samp.description.html` — A description of the application, in HTML.
`samp.icon.url` — The URL of an icon in png, gif or jpeg format.
`samp.documentation.url` — The URL of a documentation web page.

All of the above are OPTIONAL, but `samp.name` is strongly RECOMMENDED. These keys form the basis of an extensible vocabulary as explained in Section 2.6.

3.7 MType Subscriptions

As outlined above, an MType is a string which defines the semantics of a message. MTypes have a hierarchical form. Their syntax is given by the following BNF:

```
<mchar>  ::= [0-9A-Za-z] | "-" | "_"
<atom>   ::= <mchar> | <atom> <mchar>
<mtype>  ::= <atom> | <mtype> "." <atom>
```

Examples might be "`samp.hub.event.shutdown`" or "`file.load`".

A client may *subscribe* to one or more MTypes to indicate which messages it is willing to receive. A client will only ever receive messages with MTypes to which it has subscribed. In order to do this it passes a subscriptions `map` to the hub. Each key of this map is an MType string to which the client wishes to subscribe, and the corresponding value is a map which may contain additional information about that subscription. Currently, no keys are defined for these per-MType maps, so typically they will be empty (have no entries). The use of a map here is to permit experimentation and perhaps future extension of the SAMP standard.

As a special case, simple wildcarding is permitted in subscriptions. The keys of the subscription map may actually be of the form `<msub>`, where

```
<msub>   ::= "*" | <mtype> "." "*"
```

Thus a subscription key "`file.event.*`" means that a client wishes to receive any messages with MType which begin "`file.event.`". This does not include "`file.event`". A subscription key "`*`" subscribes to all MTypes. Note that the wildcard "`*`" character may only appear at the end of a subscription key, and that this indicates subscription to the entire subtree.

More discussion of MTypes, including their semantics, is given in Section 6.

3.8   Message Encoding

A message is an abstract container for the information we wish to send to another application. The message itself is that data which should arrive at the receiving application. It may be transmitted along with some external items (e.g. sender, recipient and message identifiers) required to ensure proper delivery or handling.

A message is encoded for SAMP transmission as a `map` with the following REQUIRED keys:

`samp.mtype` — A `string` giving the MType which defines the meaning of the message. The MType also, via external documentation, defines the names, types and meanings of any parameters and return values. MTypes are discussed in more detail in Section 6.

`samp.params` — A `map` containing the values for the message's named parameters. These give the data required for the receiver to act on the message, for instance the URL of a given file. The names, types and semantics of these parameters are determined by the MType. Each key in this map is the name of a parameter, and the corresponding value is that parameter's value.

These keys form the basis of an extensible vocabulary as explained in Section 2.6.

3.9   Response Encoding

A response is what may be returned from a recipient to a sender giving the result of processing a message (though in the case of the Notification delivery pattern, no such response is generated or returned). It may contain MType-specific return values, or error information, or both.

A response is encoded for SAMP transmission as a `map` with the following keys:

`samp.status` (REQUIRED) — A `string` summarising the result of the processing. It may take one of the following defined values:

samp.ok: Processing successful. The `samp.result`, but not the `samp.error` entry SHOULD be present.

samp.warning: Processing partially successful. Both `samp.result` and `samp.error` entries SHOULD be present.

samp.error: Processing failed. The `samp.error`, but not the `samp.result` entry SHOULD be present.

These values form the basis of an extensible vocabulary as explained in Section 2.6.

samp.result (REQUIRED in case of full or partial success) — A `map` containing the values for the message's named return values. The names, types and semantics of these returns are determined by the MType. Each key in this map is the name of a return value, and the corresponding value is the actual value. Note that even for MTypes which define no return values, the value of this entry MUST still be a `map` (typically an empty one).

samp.error (REQUIRED in case of full or partial error) — A `map` containing error information. The following keys are defined for this map:

samp.errortxt (REQUIRED) — A short string describing what went wrong. This will typically be delivered to the user of the sender application.

samp.usertxt (OPTIONAL) — A free-form string containing any additional text an application wishes to return. This may be a more verbose error description meant to be appended to the `samp.errortxt` string, however it is undefined how this string should be handled when received.

samp.debugtxt (OPTIONAL) — A longer string which may contain more detail on what went wrong. This is typically intended for debugging purposes, and may for instance be a stack trace.

samp.code (OPTIONAL) — A string containing a numeric or textual code identifying the error, perhaps intended to be parsable by software. Values beginning "`samp.`" are reserved.

These keys form the basis of an extensible vocabulary as explained in Section 2.6.

These keys form the basis of an extensible vocabulary as explained in Section 2.6.

In most cases, such responses will be generated by a Recipient client and forwarded by the Hub to the Sender. In some cases however the hub may pass to the sender an error response it has generated itself on behalf of the recipient. In particular, if the hub determines that no response will

ever be received from the recipient (perhaps because the recipient has unregistered without replying) the hub MAY generate and forward a response with `samp.status=samp.error` and the `samp.code` key in the `samp.error` structure set to "`samp.noresponse`". Clients SHOULD NOT generate such `samp.code=samp.noresponse` responses themselves.

3.10   Sending and Receiving Messages

As outlined in Section 2.5, three messaging patterns are supported, differing according to whether and how the response is returned to the sender. For a given MType there may be a messaging pattern that is most typically used, but there is nothing in the protocol that ties a particular MType to a particular messaging pattern; any MType may legally be sent using any delivery pattern.

From the point of view of the sender, there are three ways in which a message may be sent, and from the point of view of the recipient there are two ways in which one may be received. These are described as follows.

Notification: In the notification pattern, communication is only in one direction:

1. The sender sends a message to the hub for delivery to one or more recipients.
2. The hub forwards the message to those requested recipients which are subscribed.
3. No reply from the recipients is expected or possible.

Notifications can be sent to a given recipient or broadcast to all recipients. The notification pattern for a single recipient is illustrated in Figure 3.
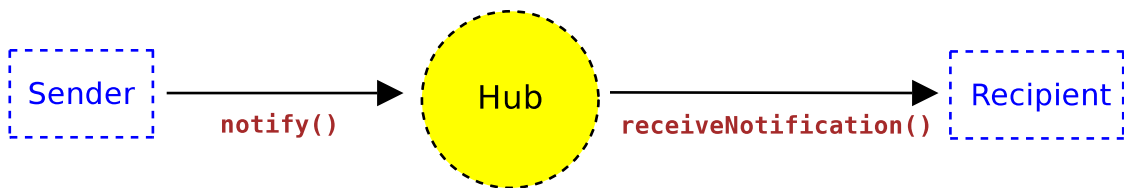

Figure 3: Notification pattern

Asynchronous Call/Response: In the asynchronous call pattern, *message tags* and *message identifiers* are used to tie together messages and their replies:

22

1. The sender sends a message to the hub for delivery to one or more recipients, supplying along with the message a tag string of its own choice, *msg-tag*. In return it receives a unique identifier string, *msg-id*.
2. The hub forwards the message to the appropriate recipients, supplying along with the message an identifier string, *msg-id*.
3. Each recipient processes the message, and sends its response back to the hub along with the ID string *msg-id*.
4. Using a callback, the hub passes the response back to the original sender along with the ID string *msg-tag*.

The sender is free to use any value for the *msg-tag*. There is no requirement on the form of the hub-generated *msg-id* (it is not intended to be parsed by the recipient), but it MUST be sufficient for the hub to pair messages with their responses reliably, and to pass the correct *msg-tag* back with the response to the sender[1]. In most cases the sender will not require the *msg-id*, since the *msg-tag* is sufficient to match calls with responses. For this reason, the sender need not retain the *msg-id* and indeed need not wait for it, avoiding a hub round trip at send time. The only case in which the sender may require the *msg-id* is if it needs to communicate later with the recipient about the message that was sent, for instance as part of a progress report. Asynchronous calls may be sent to a given recipient or broadcast to all recipients. In the latter case, the sender SHOULD be prepared to deal with multiple responses to the same call. The asynchronous pattern is illustrated in Figure 4.
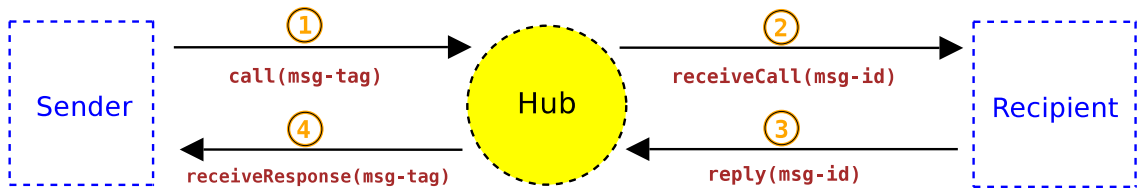
Figure 4: Asynchronous Call/Response pattern

---

[1] One way a hub might implement this is to generate *msg-id* by concatenating the sender's client ID and the *msg-tag*. When any response is received the hub can then unpack the accompanying *msg-id* to find out who the original sender was and what *msg-tag* it used. In this way the hub can determine how to pass each response back to its correct sender without needing to maintain internal state concerning messages in progress. Hub and client implementations may wish to exploit this freedom in assigning message IDs for other purposes as well, for instance to incorporate timestamps or checksums.

Synchronous Call/Response A synchronous utility method is provided
by the hub, mainly for the convenience of environments where deal-
ing with asynchronicity might be a problem. The hub will provide
synchronous behaviour to the sender, interacting with the receiver in
exactly the same way as for the asynchronous case above.

1. The sender sends a message to the hub for delivery to a given recip-
   ient, optionally specifying as well a maximum time it is prepared
   to wait. The sender's call blocks until a response is available.
2. The hub forwards the message to the recipient, supplying along
   with the message an ID string, *msg-id*.
3. The recipient processes the message, and sends its response back
   to the hub along with the ID string *msg-id*.
4. The hub passes back the response as the return value from the
   original blocking call made by the sender. If no response is re-
   ceived within the sender's specified timeout the blocking call will
   terminate with an error. The hub is not guaranteed to wait indef-
   initely; it MAY in effect impose its own timeout.

There is no broadcast counterpart for the synchronous call. This pat-
tern is illustrated in Figure 5.
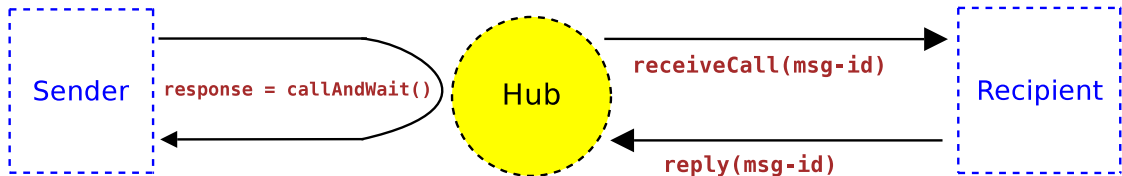
Figure 5: Synchronous Call/Response pattern

Note that the two different cases from the receiver's point of view, *No-
tification* and *Call/Response*, differ only in whether a response is returned
to the hub. In other respects the receiver SHOULD process the message in
exactly the same way for both patterns.

Although it is REQUIRED by this standard that client applications
provide a Response for every Call that they receive, there is no way that
the hub can enforce this. Senders using the Synchronous or Asynchronous
Call/Response patterns therefore should be aware that badly-behaved recipi-
ents might fail to respond, leading to calls going unanswered indefinitely. The
timeout parameter in the Synchronous Call/Response pattern provides some
protection from this eventuality; users of the Asynchronous Call/Response
pattern may or may not wish to take their own steps.

3.11   Operations a Hub Must Support

This section describes the operations that a hub MUST support and the associated data that MUST be sent and received. The precise details of how these operations map onto method names and signatures is Profile-dependent. The mapping for the Standard Profile is described in Section 4.2, and for the Web Profile in Section 5.2.2.

- `map reg-info = register()`
  Method called by a client wishing to register with the hub. The form of `reg-info` is given in Section 3.5. Note that the form of this call may vary according to the requirements of the particular Profile in use. For instance authentication tokens may be passed in one or both directions to complete registration.

- `unregister()`
  Method called by a client wishing to unregister from the hub.

- `declareMetadata(map metadata)`
  Method called by a client to declare its metadata. May be called zero or more times to update hub state; the most recent call is the one which defines the client's currently declared metadata. The form of the `metadata` map is given in Section 3.6.

- `map metadata = getMetadata(string client-id)`
  Returns the metadata information for the client whose public ID is `client-id`. The form of the `metadata` map is given in Section 3.6.

- `declareSubscriptions(map subscriptions)`
  Method called by a callable client to declare the MTypes it wishes to subscribe to. May be called zero or more times to update hub state; the most recent call is the one which defines the client's currently subscribed MTypes. The form of the `subscriptions` map is given in Section 3.7.

- `map subscriptions = getSubscriptions(string client-id)`
  Returns the subscribed MTypes for the client whose public ID is `client-id`. The form of the `subscriptions` map is given in Section 3.7.

- `list client-ids = getRegisteredClients()`
  Returns the list of public ids of all other registered clients. The caller's ID (`samp.self-id` from Section 3.5) is not included, but the hub's ID

25

(`samp.hub-id` from Section 3.5) is.

- `map client-subs = getSubscribedClients(string mtype)`
  Returns a map with an entry for all other registered clients which are subscribed to the MType `mtype`. The key for each entry is a subscribed client ID, and the value is a (possibly empty) `map` providing further information on its subscription to `mtype` as described in Section 3.7. An entry for the caller is not included, even if it is subscribed. `mtype` MUST NOT include wildcards.

- `notify(string recipient-id, map message)`
  Sends a message using the Notification pattern to a given recipient. The form of the `message` map is given in Section 3.8. An error results if the recipient is not subscribed to the message's MType.

- `list recipient-ids = notifyAll(map message)`
  Sends a message using the Notification pattern to all other clients which are subscribed to the message's MType. The form of the `message` map is given in Section 3.8. The return value is a `list` of the client IDs of the clients to which an attempt to send the message is made.

- `string msg-id = call(string recipient-id, string msg-tag, map message)`
  Sends a message using the Asynchronous Call/Response pattern to a given recipient. The form of the `message` map is given in Section 3.8. An error results if the recipient is not subscribed to the message's MType, or if the invoking client is not Callable.

- `map calls = callAll(string msg-tag, map message)`
  Sends a message using the Asynchronous Call/Response pattern to all other clients which are subscribed to the message's MType. The form of the `message` map is given in Section 3.8. The returned value is a `map` in which the keys are the client IDs of clients to which an attempt to send the message is made, and the values are the associated `msg-id` strings. An error results if the invoking client is not Callable.

- `map response = callAndWait(string recipient-id, map message, string timeout)`
  Sends a message using the Synchronous Call/Response pattern to a given recipient. The forms of the `message` and `response` maps are given in Sections 3.8 and 3.9. The `timeout` parameter is interpreted as

26

a *SAMP int* (Section 3.4) giving the maximum number of seconds the client wishes to wait. If the response takes longer than that to arrive this method SHOULD terminate anyway with an error (it MUST not return a `response` indicating error). Any response arriving from the recipient after that will be discarded. If `timeout`$<= 0$ then no artificial timeout is imposed. An error results if the recipient is not subscribed to the message's MType.

- `reply(string msg-id, map response)`
  Method called by a client to send its response to a given message. The form of the `response` map is given in Section 3.9.

Of these operations, only `callAndWait()` involves blocking communication with another client. The others SHOULD be implemented in such a way that clients can expect them to complete, and where appropriate return a value, on a timescale short compared to user response time.

3.12 Operations a Callable Client Must Support

This section lists the operations which a client MUST support in order to be classified as callable. The hub uses these operations when it wishes to pass information to a callable client. Note that callability is OPTIONAL for clients; special (Profile-dependent) steps may be required for a client to inform the hub how it can be contacted, and thus become callable. Clients which are not callable can send messages using the Notify or Synchronous Call/Response patterns, but are unable to receive messages or to use Asynchronous Call/Response, since these operations rely on client callbacks from the hub.

The precise details of how these operations map onto method names and signatures is Profile-dependent. The mapping for the Standard Profile is given in Section 4.2 and for the Web Profile in Section 5.2.5.

- `receiveNotification(string sender-id, map message)`
  Method called by the hub when dispatching a notification to its recipient. The form of the `message` map is given in Section 3.8.

- `receiveCall(string sender-id, string msg-id, map message)`
  Method called by the hub when dispatching a call to its recipient. The client MUST at some later time make a matching call to `reply()` on the hub. The form of the `message` map is given in Section 3.8.

- `receiveResponse(string responder-id, string msg-tag,`
                    `map response)`

  Method used by the hub to dispatch to the sender the response of an earlier asynchronous call. The form of the `response` map is given in Section 3.9.

3.13 Error Processing

Errors encountered by clients when processing Call/Response-pattern messages themselves (in response to a syntactically legal `receiveCall()` operation) SHOULD be signalled by returning appropriate content in the response map sent back in the matching `reply()` call, as described in Section 3.9.

In the case of failed calls of the operations defined in Sections 3.11 and 3.12, for instance syntactically invalid parameters or communications failures, hubs and clients SHOULD where possible use the usual error reporting mechanisms of the transport protocol in use.

Where it is problematic or impossible to use the transport protocol's error reporting mechanisms, in the case of a Call/Response pattern message, the hub MAY signal errors by generating and passing back to the sender a suitable response map as described in Section 3.9.

4 Standard Profile

Section 3 provides an abstract definition of the operations and data structures used for SAMP messaging. As explained in Section 2.7, in order to implement this architecture some concrete choices about how to instantiate these concepts are required.

This section gives the details of a SAMP Profile based on the XML-RPC specification [7]. Hub discovery is via a lockfile in the user's home directory.

XML-RPC is a simple general purpose Remote Procedure Call protocol based on sending XML documents using HTTP POST (it resembles a very lightweight version of SOAP). Since the mappings from SAMP concepts such as API calls and data types to their XML-RPC equivalents is very straightforward, it is easy for application authors to write compliant code without use of any SAMP-specific library code. An XML-RPC library, while not essential, will make coding much easier; such libraries are available for many languages.

4.1 Data Type Mappings

The SAMP argument and return value data types described in Section 3.3 map straightforwardly onto XML-RPC data types as follows:

SAMP type		XML-RPC element
string	—	`<string>`
list	—	`<array>`
map	—	`<struct>`

The `<value>` children of `<array>` and `<struct>` elements themselves contain children of type `<string>`, `<array>` or `<struct>`.

Note that other XML-RPC scalar types (`<i4>`, `<double>` etc) are not used; even where the semantic sense of a value matches one of those types it MUST be encoded as an XML-RPC `<string>`.

4.2 API Mappings

The operation names in the SAMP hub and client abstract APIs (Sections 3.11 and 3.12) very nearly have a one to one mapping with those in the Standard Profile XML-RPC APIs. The Standard Profile API MUST be implemented as described in Sections 3.11 and 3.12 with the following RE-QUIRED adjustments:

1. The XML-RPC method names (i.e. the contents of the XML-RPC `<methodName>` elements) are formed by prefixing the hub and client abstract API operation names with "`samp.hub.`" or "`samp.client.`" respectively.

2. The `register()` operation takes the following form:

   - `map reg-info = register(string samp-secret)`

   The argument is the `samp-secret` value read from the lockfile (see Section 4.3). The returned `reg-info` map contains an additional entry with key `samp.private-key` whose value is a string generated by the hub.

3. *All* other hub and client methods take the `private-key` as their first argument.

4. A new method, `setXmlrpcCallback()` is added to the hub API.

   - `setXmlrpcCallback(string private-key, string url)`

   This informs the hub of the XML-RPC endpoint on which the client is listening for calls from the hub. The client is not considered Callable unless and until it has invoked this method.

5. Another new method, `ping()` is added to the hub API. This may be called by registered or unregistered applications (as a special case the `private-key` argument may be omitted), and can be used to determine whether the hub is responding to requests. Any non-error return indicates that the hub is running.

The `private-key` string referred to above serves two purposes. First it identifies the client in hub/client communications. Some such identifier is required, since XML-RPC calls have no other way of determining the sender's identity. Second, it prevents application spoofing, since the private key is never revealed to other applications, so that one application cannot pose as another in making calls to the hub.

The usual XML-RPC fault mechanism is used to respond to invalid calls as described in Section 3.13. The XML-RPC `<fault>`'s `<faultString>` element SHOULD contain a user-directed message as appropriate and the `<faultCode>` value has no particular significance.

4.3 Lockfile and Hub Discovery

Hub discovery is performed by examining a lockfile to determine hub connection parameters, specifically the XML-RPC endpoint at which the hub can be found, and a "secret" token which affords some measure of security, given suitable restrictions on the lockfile's readability (see Section 4.3.2). To discover the hub, a client must therefore:

1. Determine where to find the lockfile (4.3.1)
2. Read the lockfile to obtain the hub connection parameters (4.3.3)

4.3.1 Lockfile Location

The default location of the lockfile is the file named ".`samp`" in the user's home directory. However the content of the environment variable named SAMP_HUB can be used to override this default.

The value of the SAMP_HUB environment variable is of the form `<samphub-value>`, as defined by the following BNF production:

```
<samphub-value>  ::=  <hub-location>
<hub-location>   ::=  <stdlock-prefix> <stdlock-url>
<lockurl-prefix> ::=  "std-lockurl:"
<stdlock-url>    ::=  (any URL)
```

The `<stdlock-url>` will typically, but not necessarily, be a file-type URL (as described in RFC 1738, section 3.10 [9]). So for instance to indicate that the lockfile to be used will be the file "`/tmp/samp1`", you would set

```
SAMP_HUB=std-lockurl:file:///tmp/samp1
```

Although no other form of the `<hub-location>` value is defined here, the intention is that the SAMP_HUB environment variable MAY be used with prefixes other than "`std-lockurl:`" to indicate use of other, non-Standard, profiles. Issues may in future arise related to the need to indicate multiple profiles or profile variants at once; the impact of this requirement on the syntax and semantics of the SAMP_HUB variable is for now deferred.

To locate the lockfile therefore, a Standard Profile-compliant client MUST determine whether an environment variable named SAMP_HUB exists; if so, the client MUST examine the variable's value; if the value begins with the prefix "`std-lockurl:`" the client MUST interpret the remainder of the value as a URL whose content is the text of the lockfile to be used for hub discovery. If no SAMP_HUB environment variable exists, the client MUST use the file ".samp" in the user's home directory as the lockfile to be used for hub discovery. If the variable exists, but its value begins with a different prefix, the client MAY interpret that in some non-Standard way for hub discovery.

Rules for a Standard Profile-compliant hub to use when writing lockfiles are similar, but if a hub is unable or unwilling to write a lockfile such that it can be read using the above procedure, it MUST signal an error at the startup and then abort. For practical reasons, a hub will probably only be able to write a lockfile indicated by a `file`-type URL, not for instance an arbitrary `http`-type one. Lockfiles SHOULD be created with appropriate access restrictions as discussed in Section 4.3.2.

The existence or readability of the lockfile MAY be taken (e.g. by a hub deciding whether to start or not) to indicate that a hub is running. However it is RECOMMENDED to attempt to contact the hub at the given XML-RPC URL (e.g. by calling `ping()`) to determine whether it is actually alive.

The "home directory" referred to above is a somewhat system-dependent concept: we define it as the value of the `HOME` environment variable on Unix-like systems and as the value of the `USERPROFILE` environment variable on Microsoft Windows[2]. "Environment variable" is itself potentially a system-dependent concept, but it is clear how to interpret it for all platforms on which we currently expect SAMP to be used, so no further explanation is provided here.

In version 1.11 of the standard, the lockfile was always in the ".samp" file in the user's home directory. The option of setting the SAMP_HUB environ-

---

[2] Note to Java developers: contrary to what you might expect, the `user.home` system property on Windows does *not* give you the value of `USERPROFILE`. See `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4787931`.

ment variable to override this has been introduced to allow more flexibility; for instance one user can run multiple unconnected hubs, or multiple users can share the same hub. If no SAMP_HUB environment variable is defined, client and hub behaviour is exactly as in version 1.11.

4.3.2 Security Considerations

The hub SHOULD normally create the lockfile with file permissions which allow only its owner to read it. This provides a measure of security in that only processes with the same privileges as the hub process (hence presumably running under the same user ID) will be able to register with the hub, since only they will be able to provide the secret token, obtained from the lockfile, which is required for registration. Thus under normal circumstances all Standard Profile clients can be presumed to be running with the same level of trust, so that no security issues of the type discussed in Section 2.8 arise.

If the lockfile is made available in some way other than an owner-only readable file, for instance via an unprotected `http`-type URL in order to facilitate use of the same hub by multiple users on different hosts, there is a potential security risk. In that case, protection through an authentication and/or authorization mechanism might be adopted by the hub implementations, for instance exploiting the TLS cryptographic protocol [10].

4.3.3 Lockfile Content

The format of the lockfile is given by the following BNF productions:

```
<file>          ::=   <lines>
<lines>         ::=   <line> | <lines> <line>
<line>          ::=   <line-content> <EOL> | <EOL>
<line-content>  ::=   <comment> | <assignment>
<comment>       ::=   "#" <any-string>
<assignment>    ::=   <name> "=" <any-string>
<name>          ::=   <token-string>
<token-string>  ::=   <token-char> | <token-string> <token-char>
<any-string>    ::=   <any-char> | <any-string> <any-char>
<EOL>           ::=   "\r" | "\n" | "\r" "\n"
<token-char>    ::=   [a-zA-Z0-9] | "-" | "_" | "."
<any-char>      ::=   [\x20-\x7f]
```

The only parts which are significant to SAMP clients/hubs are (a) existence of the file and (b) `<assignment>` lines.

A legal lockfile MUST provide (in any order) unique assignments for the following tokens:

`samp.secret` — An opaque text string which must be passed to the hub to permit registration.

`samp.hub.xmlrpc.url` — The XML-RPC endpoint for communication with the hub.

`samp.profile.version` — The version of the SAMP Standard Profile implemented by the hub ("1.3" for the version described by this document).

These keys form the basis of an extensible vocabulary as explained in Section 2.6. Other blank, comment or assignment lines may be included as desired.

An example lockfile might therefore look like this:

```
# SAMP lockfile written 2011-12-22T05:30:01
# Required keys:
samp.secret=734144fdaab8400a1ec2
samp.hub.xmlrpc.url=http://andromeda.star.bris.ac.uk:8001/xmlrpc
samp.profile.version=1.3
# Info stored by hub for some private reason:
com.yoyodyne.hubid=c80995f1
```

4.3.4 Hub Discovery Sequences

The hub discovery sequences are therefore as follows:

- Client startup:
    - Determine hub existence as above
    - If no hub, client MAY start its own hub
    - Acquire `samp.secret` value from lockfile
    - If pre-existing or own hub is running, call `register()` and zero or more of `setXmlrpcCallback()`, `declareMetadata()`, `declareSubscriptions()`

- Hub startup:
    - Determine hub existence as above
    - If hub is running, exit
    - Otherwise, start up XML-RPC server
    - Write lockfile containing mandatory assignments including XML-RPC endpoint, using appropriate access restrictions

- Hub shutdown:

– Remove lockfile (it is RECOMMENDED to first check that this is the lockfile written by self)
  – Notify candidate clients that shutdown will occur
  – Shut down services

A hub implementation SHOULD make its best effort to perform the shutdown sequence above even if it terminates as a result of some error condition.

Note that manipulation of a file is not atomic, so that race conditions are possible. For instance a client or hub examining the lockfile may read it after it has been created but before it has been populated with the mandatory assignments, or two hubs may look for a lockfile simultaneously, not find one, and both decide that they should therefore start up, one presumably overwriting the other's lockfile. Hub and client implementations should be aware of such possibilities, but may not be able to guarantee to avoid them or their consequences. In general this is the sort of risk that SAMP and its Standard Profile are prepared to take — an eventuality which will occur sufficiently infrequently that it is not worth significant additional complexity to avoid. In the worst case a SAMP session may fail in some way, and will have to be restarted manually.

4.4   Examples

Here is an example in pseudo-code of how an application might locate and register with a hub, and send a message requiring no response to other registered clients.

```
# Locate and read the lockfile.
string hubvar-value = readEnvironmentVariable("SAMP_HUB");
string lock-location = getLockfileLocation(hubvar-value);
map lock-info = readLockfile(lock-location);

# Extract information from lockfile to locate and register with hub.
string hub-url = lock-info.getValue("samp.hub.xmlprc.url");
string samp-secret = lock-info.getValue("samp.secret");

# Establish XML-RPC connection with hub
# (uses some generic XML-RPC library)
xmlrpcServer hub = xmlrpcConnect(hub-url);

# Register with hub.
map reg-info = hub.xmlrpcCall("samp.hub.register", samp-secret);
string private-key = reg-info.getValue("samp.private-key");
```

```
# Store metadata in hub for use by other applications.
map metadata = ("samp.name" -> "dummy",
               "samp.description.text" -> "Test Application",
               "dummy.version" -> "0.1-3");
hub.xmlrpcCall("samp.hub.declareMetadata", private-key, metadata);

# Send a message requesting file load to all other
# registered clients, not wanting any response.
map loadParams = ("filename" -> "/tmp/foo.bar");
map loadMsg = ("samp.mtype" -> "file.load",
               "samp.params" -> loadParams);
hub.xmlrpcCall("samp.hub.notifyAll", private-key, loadMsg);

# Unregister
hub.xmlrpcCall("samp.hub.unregister", private-key);
```

The first few XML-RPC documents sent over the wire for this exchange
would look something like the following. The registration call from the client
to the hub:

```
POST /xmlrpc HTTP/1.0
User-Agent: Java/1.5.0_10
Content-Type: text/xml
Content-Length: 189

<?xml version="1.0"?>
<methodCall>
  <methodName>samp.hub.register</methodName>
  <params>
    <param><value><string>734144fdaab8400a1ec2</string></value></param>
  </params>
</methodCall>
```

which leads to the response:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: text/xml
Content-Length: 464

<?xml version="1.0"?>
<methodResponse>
  <params><param><value><struct>
    <member>
      <name>samp.private-key</name>
```

```
      <value><string>client-key:1a52fdf</string></value>
    </member>
    <member>
      <name>samp.hub-id</name>
      <value><string>client-id:0</string></value>
    </member>
    <member>
      <name>samp.self-id</name>
      <value><string>client-id:4</string></value>
    </member>
  </struct></value></param></params>
</methodResponse>
```

The client can then declare its metadata: the response to this call has no useful content so can be ignored or discarded.

```
POST /xmlrpc HTTP/1.0
User-Agent: Java/1.5.0_10
Content-Type: text/xml
Content-Length: 600

<?xml version="1.0"?>
<methodCall>
  <methodName>samp.hub.declareMetadata</methodName>
  <params>
    <param><value><string>app-id:1a52fdf-2</string></value></param>
    <param><value><struct>
      <member>
        <name>samp.name</name>
        <value><string>dummy</string></value>
      </member>
      <member>
        <name>samp.description.text</name>
        <value><string>Test application</string></value>
      </member>
      <member>
        <name>dummy.version</name>
        <value><string>0.1-3</string></value>
      </member>
    </struct></value></param>
  </params>
</methodCall>
```

The message itself is sent from the client to the hub as follows:

```
POST /xmlrpc HTTP/1.0
User-Agent: Java/1.5.0_10
Content-Type: text/xml
Content-Length: 523

<?xml version="1.0"?>
<methodCall>
  <methodName>samp.hub.notifyAll</methodName>
  <params>
    <param><value><string>app-id:1a52fdf-2</string></value></param>
    <param><value><struct>
      <member>
        <name>samp.mtype</name>
        <value>file.load</value>
      </member>
      <member>
        <name>samp.params</name>
        <value><struct>
          <name>filename</name>
          <value>/tmp/foo.bar</value>
        </struct></value>
      </member>
    </struct></value></param>
  </params>
</methodCall>
```

Again, there is no interesting response.

5   Web Profile

This section defines the SAMP Web Profile which allows web applications to communicate with a SAMP hub. A *web application* in this context is code which is downloaded by a web browser from a remote server, usually as part of a web page, and which then runs from within that browser. The most common platforms (browser-based runtime environments) for such applications are currently JavaScript (a.k.a. JScript, ECMAScript), Java applets, Adobe Flash, and Microsoft Silverlight. For security reasons, these runtime environments run the web applications that they host inside a secure "sandbox", which imposes restrictions on access to resources, making it impossible to use the Standard Profile defined in Section 4. Java applets provide a client-controlled cross-browser mechanism, based on code signing, for circumventing these restrictions, but the others do not.

Section 5.1 gives an illustrative overview of the way the Web Profile achieves its communication requirements, with comparison to the Standard Profile. Section 5.2 describes in detail how the Web Profile hub is implemented in order to provide the functionality defined by the SAMP abstract hub and client APIs (Sections 3.11 and 3.12). Section 5.3 outlines the steps that a Web Profile client must take to locate and communicate with the hub. The important topic of the security implications of this scheme, and measures which hub implementations can take in view of these, is covered separately in Section 5.4.

5.1 Overview and Comparison with Standard Profile

The Web Profile is based on the Standard Profile (Section 4), but with some modifications which allow clients to overcome the restrictions imposed by the browser sandbox.

Browser restrictions present four main problems for a web-based SAMP client: hub discovery, outward hub communication, inward hub communication and use of third-party URLs. These are solved in the Web Profile by use of a well-known port, use of standard and de facto cross-origin access techniques, reversed HTTP communication, and URL proxying. These solutions are described, with comparison to the approaches used by the Standard Profile, in the following subsections.

5.1.1 Hub Discovery

A Standard Profile client locates the hub by reading a "lockfile" at a well-known location in the filesystem, which provides the HTTP endpoint at which the hub XML-RPC server is listening and a token which the client must present in order to register. Web applications have no access to the local filesystem and so are unable to read such a lockfile.

In the Web profile, the hub HTTP server listens instead on a well-known port on the local host. The hub will apply some security measures at registration time (Section 5.4.2), but they are not based on presentation of a secret token.

Note that since this well-known port number is fixed, it is not possible for more than one Web Profile hub to run on the same host. The Web Profile Hub and corresponding web browser MUST run on the same host, and SHOULD always be run by the same user.

For a web client to be able to access this well-known port at all, the cross-origin techniques discussed in the next section are required.

5.1.2 Outward Communications

In the Standard Profile, all hub communication is done using the HTTP-based XML-RPC protocol [7], usually to a port on the local host.

This is problematic for web-based clients, since so-called "cross-origin" or "cross-domain" policies enforced by browsers restrict HTTP access under normal circumstances so that web applications may *only* make HTTP requests to URLs at their own *Origin* [18], that is to URLs on the server from which the web application itself was downloaded. This deliberately excludes access to a server on the local host, which is where the SAMP hub is likely to reside.

Since cross-origin access is a common requirement for web-based clients, and it is not always in conflict with the security concerns of servers, a number of platform-dependent but widely-used mechanisms have been implemented in browser technology which allow a sandboxed client to talk to an HTTP server which has explicitly opted in for such cross-origin communications. A Web Profile hub will implement one or more of these cross-origin workarounds (Section 5.2.3) and so permit Web Profile clients running in the relevant browser runtime environment(s) to make HTTP requests to itself, thereby allowing client-to-hub XML-RPC calls.

5.1.3 Inward Communications

If it wishes to receive as well as send messages, and also to make asynchronous calls, a SAMP client must declare itself *Callable*, by providing the Hub with a profile-dependent means to invoke the client API defined in Section 3.12.

In the Standard Profile a client declares itself Callable by providing to the Hub an HTTP endpoint to which the Hub may make XML-RPC requests. Thus, the client must itself run a publicly accessible HTTP server in order to be callable. Running an HTTP server is typically not within the capabilities of a web application.

In the Web Profile, hub-to-client communication is effected by reversing the direction of the XML-RPC calls, and hence of the HTTP requests. Instead of the client running a server which listens for incoming messages from the Hub, the Hub maintains a queue of messages destined for the client, and the client polls the Hub to find out if any are available. The client may either make periodic short-timeout requests to the hub, or make a long-timeout ("long poll") request which will return early if and when one or more messages are available. This effects inward communications using only the same outward HTTP capability discussed in the previous section.

39

5.1.4 Third-Party URLs

Although it is not fundamental to the SAMP protocol itself, many SAMP MTypes are defined in such a way that a receiving client must retrieve data from a URL external to the SAMP client-hub system in order to act on them. For instance the `table.load.votable` MType has an argument named "`url`", whose value is the location of the VOTable document to be loaded. Such URLs may point to the local filesystem, to a server run by the sending client, or to some other web server internal or external to the host on which the SAMP communications are taking place. Similar considerations apply to some of the client metadata items (Section 3.6), for instance `samp.icon.url`. In any of these cases, it is likely that a browser-based client will be blocked by the browser's cross-origin policy from accessing the content of the resource in question.

The Web Profile therefore mandates that the Hub must provide to registered clients a mechanism for translating arbitrary URLs into cross-origin-accessible URLs with the same content as the specified resource. Since a hub must already be providing a cross-origin capable HTTP service accessible from the web client, it can use the same mechanism to operate a service which proxies external resources in a cross-origin capable way.

5.2 Hub Behaviour

This section specifies in detail the services that a SAMP hub must provide in order to implement the SAMP Web Profile.

The Web Profile is based on client-to-hub XML-RPC calls, with the hub residing at a well-known port, and some special measures for allowing cross-origin requests. In most ways it resembles the Standard Profile (Section 4), but there are some differences.

5.2.1 Data Type Mappings

SAMP argument and return value data types are encoded into XML-RPC exactly as for the Standard Profile (Section 4.1).

5.2.2 API Mappings

The operation names in the SAMP hub API very nearly have a one to one mapping with those in the Web Profile XML-RPC API. The Web Profile Hub API MUST be implemented as described in Section 3.11, with a number of REQUIRED adjustments. These are summarised as follows, and described in more detail later.

1. The XML-RPC method names (i.e. the contents of the XML-RPC `<methodName>` elements) are formed by prefixing the hub abstract API operation names with "`samp.webhub.`". For brevity, this prefix is not written in the rest of this document, but it is to be understood on all hub API XML-RPC calls.

2. The `register` operation takes the following form (Section 5.2.4):

   - `map reg-info = register(map identity-info)`

   The `identity-info` is a map containing at least a declared application name supplied by the registering application to indicate its identity.

3. The `reg-info` map returned from the `register` method MUST contain two entries additional to those mandated by the hub API (Section 5.2.4):

   `samp.private-key:` used as the first argument of all hub API XML-RPC calls

   `samp.url-translator:` used for translation of foreign URLs for cross-origin accessibility

4. *All* hub methods other than `register` take the `private-key` as their first argument, except where otherwise noted (`ping`). For brevity, this argument is not written in the rest of this document, but it is to be understood on all hub API calls.

5. Two new methods are added to the hub API to support reversed callbacks (Section 5.2.5):

   - `allowReverseCallbacks(string allow)`
   - `map pullCallbacks(string timeout)`

6. Another new method is added to the hub API:

   - `ping()`

   This may be called by registered or unregistered applications (as a special case the `private-key` argument may be omitted), and can be used to determine whether the hub is responding to requests. Any non-error return indicates that the hub is running.

5.2.3 Hub HTTP Server

Communications are XML-RPC calls [7] from the client to the Hub. XML-RPC works using POSTs to an HTTP server. The Web Profile hub HTTP server resides on the well-known port 21012, so that clients know where to find it on the local host. The XML-RPC endpoint for Web Profile requests is at the root of that server, so that web clients can access it by POSTing to the URL "`http://localhost:21012/`".

41

In general, web applications operate inside a browser-enforced sandbox that prevents them from accessing cross-origin resources, including HTTP-based ones served from the local host. However there are a number of ways in which an HTTP server can elect to permit access from browser-based clients. In order to be useful a Web Profile hub must implement at least one of these "cross-origin workarounds".

The following cross-origin workarounds are known to exist, and can be considered for use by Web Profile hub HTTP servers:

Cross-Origin Resource Sharing: CORS [11] is a W3C standard which works by manipulation of the HTTP Origin header and related headers by the browser runtime environment and the HTTP server, allowing the HTTP server to grant cross-domain access from clients with some or all Origins. CORS forms part of the XmlHttpRequest Level 2 standard [12], which is implemented by, at least, Chrome v2.0+, Firefox v3.5+ and Safari v4.0+. Microsoft's IE8+ implements CORS via its own non-standard XDomainRequest object. This standard belongs to the loose HTML5 family of technologies, and it is likely that support will become wider in the future. A Web Profile hub HTTP server can grant unrestricted access to CORS-aware web applications by following the instructions in the CORS standard to enable both *simple* and *preflight* requests from clients with any Origin.

Flash cross-domain policy: Adobe's Flash browser plugin makes use of a resource named "`crossdomain.xml`", which, if present on an external HTTP server, is taken to indicate willingness to serve cross-domain requests [13]. This has emerged as something of a de facto standard, and the crossdomain file is honoured by Silverlight and unsigned Java Applets/WebStart applications[3] as well as for Flash applications. A Web Profile hub HTTP server can grant unrestricted access to Flash-like web applications by serving a resource named "`/crossdomain.xml`" with a Content-Type header of "`text/x-cross-domain-policy`" and content like:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy
        SYSTEM "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="all"/>
  <allow-access-from domain="*"/>
  <allow-http-request-headers-from domain="*" headers="*"/>
</cross-domain-policy>
```

Silverlight cross-domain policy: Microsoft's Silverlight environment will take note of Flash-style `crossdomain.xml` files, so the above measure ought to permit Silverlight clients to access a compliant HTTP

---

[3]Support for the crossdomain.xml file is reportedly implemented in Java v1.6.0_10 and later, see `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6676256`.

server. However, Silverlight has its own cross-domain policy mechanism [14], which may be implemented in addition. A Web Profile hub HTTP server can grant unrestricted access to Silverlight web applications by serving a resource named "/clientaccesspolicy.xml" with a Content-Type header of "text/xml" and content like:

```
<?xml version="1.0"?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from>
        <domain uri="http://*"/>
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true"/>
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

If the hub implements these cross-origin workarounds it is believed that cross-origin access, hence Web Profile SAMP access, can be provided from nearly all browsers. Most modern browsers support CORS for JavaScript, nearly all others support Flash, and it is possible for JavaScript applications to make use of Flash libraries for their SAMP communications[4]. Maximum interoperability therefore can be achieved by implementing all of these, or at least CORS and Flash, in the Web Profile HTTP server. There are however security implications of which ones to implement, discussed in Section 5.4.2.2.

In the usual browser-hub configuration, web applications will always seek the Web Profile HTTP server on the local host. Since no legitimate use of the Web Profile HTTP server is expected from non-local hosts, it is therefore strongly RECOMMENDED for security reasons that the Web Profile HTTP server refuses HTTP requests from external hosts with a 403 Forbidden status. This recommendation and possible exceptions to it are discussed further in Section 5.4.2.1.

5.2.4 Registration

In order to request registration with the Web Profile, a client needs to invoke the following XML-RPC method:

```
map register(map identity-info)
```

The identity-info map provides information identifying the registering application which can inform the hub's decision about whether to allow registration. It has the following REQUIRED entry:

---

[4]See for instance the flXHR library at http://flxhr.flensed.com/.

samp.name — A `string` giving the name of the application wishing to register, in a form that can be presented to the user. This SHOULD be the same as the value of the `samp.name` key in the application metadata as described in Section 3.6.

Particular implementations or future versions of this standard may specify additional required or optional entries to this map.

The hub will accept or reject the registration based on the contents of the `identity-info` map, available information from the HTTP connection carrying the XML-RPC call, user confirmation, and the hub's own security policy, as discussed in 5.4.2. The `register` XML-RPC request will not return until the hub has decided whether to accept registration. This decision may involve user interaction and hence take a significant amount of time. The likely timescales mean that an HTTP timeout is possible but not very probable; in case of a timeout, registration fails.

If registration is accepted, the hub MUST return to the client a SAMP map containing the entries mandated by Section 3.5 and also the following entries:

samp.private-key: The value of this key is a string which identifies the registered client. This string SHOULD be difficult for third parties to guess. This arrangement is the same as for the Standard Profile (Section 4.2)

samp.url-translator: The value of this key is a string which forms the base for a URL proxying service, used as described in Section 5.2.6

If registration is rejected, the hub MUST return to the client an XML-RPC Fault, which SHOULD have a suitably explanatory `faultString`.

5.2.5 Callable Clients

In order to be able to receive communications (incoming messages and asynchronous call replies) *from* the hub, the Web Profile provides for the client to be able to poll the hub server for any messages or replies which are ready for receipt. In this way, such communications are pulled by the client rather than being pushed by the hub, so that no server component is required on the client side.

Two hub methods are provided to implement this:

- `allowReverseCallbacks(string allow)`
- `list pullCallbacks(string timeout-secs)`

Both these methods, like the others in the interface, are named with the `samp.webhub.` prefix and take the `private-key` as an additional first argument.

The `allow` argument of `allowReverseCallbacks` is a *SAMP boolean* ("0" for false or "1" for true), and the `timeout-secs` argument of `pullCallbacks` is a *SAMP int* (see Section 3.4).

If a client intends at some time in the future to poll for callbacks it MUST invoke `allowReverseCallbacks` with a true argument. If at some later point it decides that it will remain registered but will never poll for callbacks again it SHOULD invoke `allowReverseCallbacks` with a false argument (most clients will never make this second call). The client becomes *Callable* only when it has invoked this method with a true argument.

Having invoked `allowReverseCallbacks` with a true argument, the client SHOULD periodically invoke `pullCallbacks` whose return value gives the details of any callbacks ready for dispatch to the client. The `timeout-secs` parameter is the maximum number of seconds the client wishes to wait for a response. When the method is called, the hub SHOULD wait until at least one callback is available, and at that point SHOULD return any pending callbacks. If the elapsed time since `pullCallbacks` was received exceeds the number of seconds given by the `timeout-secs` argument, the hub SHOULD return with an empty list of callbacks. A client may therefore make a non-waiting poll by using a `timeout-secs` argument of 0. The hub MAY return with an empty list of callbacks before the given timeout has elapsed, for instance if it reaches an internal timeout limit.

The hub MAY discard pending messages before they have been polled for by the client, for instance to avoid excessive usage of resources to store them. If a `receiveCall` for an Asynchronous Call/Response-pattern message is discarded in this way, the hub SHOULD inform the sender by passing back a `samp.code=samp.noresponse`-type error response, as described in Section 3.9.

The format of the returned value from `pullCallbacks` is a `list` of elements each of which is a `map` representing a callback corresponding to one of the methods in the SAMP client API (Section 3.12). Each of these callbacks is encoded as a `map` with the following REQUIRED keys:

`samp.methodName` — The client API method name for the callback. Its value is a `string` taking one of the values "`receiveNotification`", "`receiveCall`" or "`receiveResponse`".

`samp.params` — A `list` of the parameters taken by the client API method in question, as documented in Section 3.12.

These items correspond to the elements present in an XML-RPC call.

Here is an example of a call to `pullCallbacks`. The client POSTs an XML-RPC call which requests any callbacks which are currently pending or which become available during the next 600 seconds:

```
POST /
Host: localhost:21012
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11)
 Gecko/20101028 Red Hat/3.6-2.el5 Firefox/3.6.11
Referer: \protect\vrule width0pt\protect\href{http://www.star.bris.ac.uk/\string~mbt/w
Content-Length: 284
Content-Type: text/plain; charset=UTF-8
Origin: \protect\vrule width0pt\protect\href{http://www.star.bris.ac.uk}{http://www.st

<?xml version='1.0'?>
<methodCall>
  <methodName>samp.webhub.pullCallbacks</methodName>
  <params>
    <param>
      <value><string>wk:1_fjlyrdtwtigfqhnwkqokqpbq</string></value>
    </param>
    <param>
      <value><string>600</string></value>
    </param>
  </params>
</methodCall>
```

The response, which is returned by the hub after some delay between 0 and 600 seconds, specifies a `receiveCall` operation that the client should respond to:

```
200 OK
Content-Length: 1444
Content-Type: text/xml
Access-Control-Allow-Origin: \protect\vrule width0pt\protect\href{http://www.star.bris

<?xml version='1.0' encoding='UTF-8'?>
<methodResponse>
  <params>
    <param>
      <value>
        <array>
          <data>
            <value>
              <struct>
```

46

```
          <member>
            <name>samp.methodName</name>
            <value>samp.webclient.receiveCall</value>
          </member>
          <member>
            <name>samp.params</name>
            <value>
              <array>
                <data>
                  <value>hub</value>
                  <value>hub_A_cc55_Ping-tag</value>
                  <value>
                    <struct>
                      <member>
                        <name>samp.mtype</name>
                        <value>samp.app.ping</value>
                      </member>
                      <member>
                        <name>samp.params</name>
                        <value>
                          <struct>
                          </struct>
                        </value>
                      </member>
                    </struct>
                  </value>
                </data>
              </array>
            </value>
          </member>
        </struct>
      </value>
    </data>
  </array>
  </value>
    </param>
  </params>
</methodResponse>
```

Some of the HTTP headers in the outgoing request in this example
have been added outside of the client's control by the browser runtime
environment. In particular the `Origin` inserted by the browser, and the
`Access-Control-Allow-Origin` provided in response by the Hub, indicate
that CORS negotiation [11] is in operation here to allow cross-origin access.

5.2.6 URL Translation

In order that sandboxed clients are able to obtain the content of URLs from foreign domains, as is often required by SAMP interoperation, the hub provides a service which is able to dereference general URLs.

At registration time, as described in Section 5.2.4, one of the values provided to the registering client is that of the `samp.url-translator` key. This is a partial URL which, when another URL *u1* is appended to it, will return the same content as *u1* from an HTTP GET request. If *u1* is a syntactically legal URL according to RFC 2396 [15], no additional encoding needs to be performed on it by the client prior to the concatenation.

A sample of ECMAScript code using this facility might look something like this:

```
var url_trans = reg_info["samp.url-translator"];
var u1 = msg["samp.params"]["url"];    // base URL received from message
var u2 = url_trans + u1;               // URL ready for retrieval
```

The partial translator URL might typically be implemented as a URL pointing to the same HTTP server in which the hub is hosted, with an empty query part. The content of URLs accessed in this way SHOULD be available under the same cross-origin arrangements described in Section 5.2.3. For security reasons the hub SHOULD ensure that this facility can only be used by registered clients, for instance by embedding the private key in the URL. Thus a translator URL might look something like

http://localhost:21012/translator/*client-private-key*?

The URL translation service SHOULD in general write an HTTP response with HTTP headers appropriate for the resource being served, in accordance with the HTTP version in use (e.g. [16]). Where the content type of a resource is not known (which is typical if that resource is backed by a file rather than an HTTP URI) the HTTP Content-Type header MAY be omitted.

For security reasons, such a hub URL translation service MAY refuse access to certain resources, as discussed in Section 5.4.3.2.

5.3 Client Behaviour

The steps that a client must take to register with a Web Profile hub and participate in two-way SAMP communications are as follows:

1. Prepare to make XML-RPC communications with the XML-RPC endpoint `http://localhost:21012/`. Web applications will need to do

this using a client which supports one of the cross-origin workarounds described in Section 5.2.3 and supported by the Web Profile hub.

2. Call the `register` XML-RPC method supplying a short application name and possibly other information in the `identity-info` argument. If this succeeds (returns a non-Fault XML-RPC response), the client is registered.

3. If the client wishes to receive as well as send communications (to be *Callable*), first call `allowReverseCallbacks` and then periodically call `pullCallbacks`. Call `declareSubscriptions` as required.

4. Act on retrieved callbacks as required. If any MType argument or return value is a URL, prefix it with the value of the `samp.url-translator` entry from the registration map before dereferencing it.

5. Send SAMP messages etc as required.

6. Unregister when no further SAMP activity is required, either because the user requests disconnection or on page unload or a similar event.

5.4 Security Considerations

Web browsers implement cross-origin access restrictions in order to prevent web applications from activity on a local host which presents a security risk, for instance reading and writing local files. This means that, at least in principle, a user can visit a web page without worrying about security issues, in a way which is not the case if they download and install an application to run outside a browser.

The Web Profile described in the preceding subsections however relies on neutralising these security measures to some extent. Although it only affects access to a single resource, the HTTP server on which the Web Profile hub resides, it is potentially serious since the services provided by the hub can expose sensitive resources.

Section 5.4.1 below presents an analysis of the risks, Sections 5.4.2 and 5.4.3 outline how they may be mitigated, and Section 5.4.4 summarises the security status of Web Profile hub deployments in practice.

5.4.1 Risk Analysis

Implementation in the Web Profile of one or more of the sandbox-defeating cross-origin workarounds described in Section 5.2.3 allows an untrusted, hence potentially hostile, web application to make HTTP requests to the Web Profile SAMP hub HTTP server. In the first instance, there is only one potentially sensitive action that this access permits: attempting to register with the SAMP hub. If the registration attempt is denied, the web application can

perform no useful or potentially dangerous operations (except for a denial of service attack, which sandboxed web applications are capable of in any case). If the registration is granted, the client can perform two classes of sensitive actions: first, exchange SAMP messages with other clients, and second, use the hub's URL translation service to access cross-domain URLs which would normally be blocked by the browser.

In order to protect against security breaches related to the Web Profile therefore, two lines of defence may be established: first, exercise control over which web applications are permitted to register, and second, restrict the actions that registered applications are permitted to take. These options are explored in the following sections, 5.4.2 and 5.4.3 respectively.

5.4.2 Registration Restrictions

A running Web Profile implementation may receive requests to register from any web application running in a local browser, and even some clients in other categories. Since not all such applications may be trustworthy, the Web Profile SHOULD exercise careful control over which ones are permitted to register. A Web Profile implementation is permitted to make such decisions in accordance with whatever security policy it deems appropriate, but it is RECOMMENDED that at least the restrictions described in the following subsections are considered: restricting requests to the local host (Section 5.4.2.1), requiring explicit user confirmation (Section 5.4.2.2) and attempting client authentication (Section 5.4.2.3).

5.4.2.1 Local Host Restriction As strongly RECOMMENDED in Section 5.2.3, registration requests, and in fact all access to the hub HTTP server, SHOULD under normal circumstances only be permitted from the local host. This blocks registration attempts from web or non-web applications on the internet at large.

Given this restriction, the only applications which may attempt to register with a hub run by user U are therefore:

1. web applications running in a browser run by user U on the local host
2. non-web applications run by user U on the local host
3. web or non-web applications run by users other than U on the local host

Type 1 are the applications that the Web Profile is designed to serve. Type 2 are not what the Web Profile is designed for, since they could use the Standard Profile instead, but they already have user privileges so present no additional security risk. Type 3 are potentially problematic, if the host in

question is a multi-user machine, since they may result in a different user who is already able to run processes on the local host acquiring access to the hub-owner's resources (e.g. private files). In practice the User Confirmation step (Section 5.4.2.2) should serve to distinguish type 3 from legitimate (type 1) requests, and the behaviour restrictions described in Section 5.4.3 will limit any potential damage.

There may be circumstances under which it is appropriate to relax this local host restriction, for instance to enable collaboration with a known external host not capable of Standard Profile communication, such as a mobile device operated by the hub user. However, it is RECOMMENDED that Web Profile implementations at least restrict access to the local host in their default configuration, and if access is permitted to external hosts it is only by explicit user request, and to a named host or list of hosts. Opening the well-known Web Profile hub server port to the internet at large would invite denial of service and perhaps phishing attacks in which the user is exposed to unwanted SAMP registration requests.

5.4.2.2 User Confirmation It is strongly RECOMMENDED that the Hub requires explicit confirmation from the user before any Web Profile application is allowed to register. This will normally take the form of the Hub popping up a dialogue window which requires the user to click "OK" or similar for registration to proceed. An implication of this is that the Web Profile hub must have access to the same visual display on which the browser is running, which almost certainly means the hub and the browser are run by the same user.

When enquiring about authorization the hub should make clear to the user the security implications of accepting the registration request, and should also present to the user any known information about the application attempting to register. Unfortunately, little such information is guaranteed to be available. The name declared by the application as part of its registration request will be present, but the application is free to declare any name, perhaps a misleading one. Certain HTTP headers on the incoming request may also be relevant: the "Origin" header [18] will be present for requests originating from CORS, and the "Referer" header [16, section 14.36] may be provided, though its presence and reliability is dependent on the combination of browser, platform and cross-origin workaround. Note that use of non-CORS options might on some browser/plugin platforms permit faking of HTTP headers[5], so that if the Web Profile HTTP server implements one

---

[5]See for example `http://secunia.com/advisories/22467/`, which refers to a Flash version from 2006. Hopefully browsers and plugins in current use do not contain such

51

of the non-CORS options alongside CORS this may reduce the reliability of header information even from HTTP requests which (apparently) originate from CORS. These headers should therefore be used with care.

Since only the name, which may be chosen at will by the registering application, is guaranteed present, this looks on the face of it like a poor basis on which to accept or reject registration by a potentially hostile web application.

However, in practice the timing of the request presentation provides the most useful information about the identity and credibility of the request. A user will only see such a popup dialogue at the time that a web application attempts to register with SAMP. This will normally be immediately following a deliberate user browser action like opening, or clicking a "Register" button on, a web page. If the user trusts the web page he has just interacted with, he can trust the application within it, and should hence authorize registration. If the user does not trust the web page he has just interacted with, or if the popup appears at a time when no obvious action has been taken to trigger a SAMP registration, then the user should deny registration. This pattern of user interaction, requiring authorization based on the timing of actions in a browser, is both intuitive and familiar to users; for instance it is used when launching a signed Java applet or Java WebStart application.

5.4.2.3 Client Authentication

As an additional security measure it would be desirable to make a reliable identification of the author of a web application by examining an associated digital certificate, with reference to a list of trusted certificate authorities. If a certificate reliably associated with the application could be obtained, this additional information could be presented to the user or used automatically by the hub to inform the decision about whether to accept or reject the registration request.

Unfortunately however the content of the actual application is not available to the Hub at registration time, so signing the application code will not in itself help.

The Web Profile does not at present therefore make any recommendation concerning client authentication. Implementations may however wish to attempt some level of authentication, perhaps by somehow associating a certificate with the web client's URL or Origin using the HTTP (or HTTPS) request headers noted in Section 5.4.2.2, or by use of additional credentials passed in the `identity-info` map.

---

vulnerabilities, but an assurance of this is beyond the scope of this document.

52

5.4.3 Behaviour Restrictions

Given the restrictions on client registration recommended by Section 5.4.2, there is a reasonable expectation that clients registered with the Web Profile will be trustworthy. However, the possibility remains that user carelessness or some phishing-like attack might lead to registration of hostile clients, and so Web Profile implementations may additionally restrict the behaviour of registered clients. In general, a Web Profile hub implementation MAY impose such restrictions as it sees fit, based on its chosen security policy. This may lead to the inability of some Web Profile clients to perform some legitimate SAMP operations; in such cases the hub SHOULD signal that fact to the client using an appropriate error mechanism.

Restrictions may be applied as described in the following subsections: restricting the MTypes that may be sent (Section 5.4.3.1), and restricting the scope of the URL translation service (Section 5.4.3.2).

5.4.3.1 MType Restrictions The SAMP standard imposes no restriction on the semantics of MTypes, so SAMP can in principle be used to send messages which exercise the privileges available to other SAMP clients in arbitrary ways. In practice, most SAMP MTypes are fairly harmless; a typical result is loading an image into an image viewer. While hostile abuse of such a capability could be annoying, it does not consitute a serious security concern. However one might imagine an MType that intentionally or unintentionally allowed execution of arbitrary scripting operations within the context of a connected client, and hostile abuse of such a facility could easily result in theft of or damage to data, or in other serious security breaches.

With this in mind, Web Profile hub implementations MAY impose some restrictions on the MTypes that registered clients are permitted to send, via for instance some per-MType whitelisting or blacklisting mechanism. Given the open-ended nature of the MType vocabulary, a whitelisting approach may be most appropriate.

The hub MAY also restrict MTypes that Web Profile registered clients are permitted to receive, though it is harder to imagine exploits based on message receipt.

Hubs may implement such message blocking either by hiding blocked subscriptions from other clients as appropriate, or by refusing to forward messages corresponding to blocked subscriptions. In the latter case a communication failure should be signalled by responding with an XML-RPC fault.

5.4.3.2 URL Restrictions As explained in Section 5.2.6, the Web Profile provides a service for proxying arbitrary URLs, so that web clients can access data referenced by URL in SAMP messages or metadata, which sandbox-imposed cross-origin restrictions would otherwise block them from reading.

This capability is essential for worthwhile use of many common SAMP MTypes. However, it is also open to abuse, for instance a hostile client might request to read `file:///etc/passwd` or some HTTP URL on the local host or network which is restricted to local access.

Web Profile implementations therefore MAY impose such restrictions as they see fit on the use of the URL translation service provided to web clients, in order to prevent such abuse. Blocking all access to resources which are local (`file:` or `http://localhost/`) is too strict to be useful, since the URLs referenced in SAMP messages very often fall into this category.

An appropriate policy might be to proxy only URLs which a web client is known to have some legitimate SAMP-based reason to access, namely those which have previously appeared in the metadata declared by, or in a message or response originating from, some other client. In consideration of the fact that web clients might be able to provoke other clients to emit a chosen URL, or might cooperate between themselves, such a list of permitted values SHOULD be further restricted to those URLs which first appeared in a metadata or message content or response map from a trusted (i.e. non-web) client.

Since the hub in general lacks the relevant semantic knowledge there is no foolproof way to identify URLs in metadata or messages, but checking for syntactically suitable map values (e.g. `(http|https|ftp|file)://.*`) is likely to be good enough for this purpose.

Where the Web Profile implementation declines a given URL proxy request, it MUST respond with a 403 Forbidden HTTP response.

It is also RECOMMENDED that proxied HTTP access is limited to the "safe" HTTP methods GET and optionally HEAD [16, section 9.1.1], and that user credentials (cookies, authentication etc) are not propagated. Requests using unsupported HTTP methods MUST be met with a 405 Method Not Allowed response.

5.4.4 Security Summary

The basic mechanics of the Web Profile present significant security risks for a host on which it runs. This section has described how security-conscious implementations of the Profile can mitigate those risks. Following the recommendations from Section 5.4.2 on when to permit registration provides a reasonable assurance that registered clients will be trustworthy, and in par-

ticular guarantees that clients can only register with explicit authorization from a human user. Following the recommendations from Section 5.4.3 about permitted behaviour of registered clients ensures that even if a hostile client is allowed to register it is unlikely to be able to do significant damage. By combining these measures it is believed that the level of risk associated with running a Web Profile, while it would not be appropriate for instance for financial transactions, is no greater than that encountered on a regular basis by use of the web in general.

The mitigation measures are presented as (in some cases strong) REC-OMMENDations and suggestions rather than REQUIREments, in order to allow implementations to experiment with the most appropriate configurations, which may change as a result of emerging technology and common usage patterns. Such experimentation and further consideration may result in some modification of the protocol or documentation of best practice in future versions of this document or elsewhere.

6 MTypes: Message Semantics and Vocabulary

A message contains an MType string that defines the semantic meaning of the message, for example a request for another application to load a table. The concept behind the MType is similar to that of a UCD [19] in that a small vocabulary is sufficient to describe the expected range of concepts required by a messaging system within the current scope of the SAMP protocol. Developers are free to introduce new MTypes for use within applications without restriction; new MTypes intended to be used for Hub messaging or other administrative purposes within the messaging system should be discussed within the IVOA for approval as part of the SAMP standard.

6.1 The Form of an MType

MType syntax is formally defined in Section 3.7. Like a UCD, an MType is made up of *atoms*. These are not only meaningful to the developer, but form the central concept of the message. Because the capabilities one application is searching for are loosely coupled with the details of what another may provide, there is not a rigorous definition of the *behavior* that an MType must provoke in a receiver. Instead, the MType defines a specific semantic message such as "display an image", and it is up to the receiving application to determine how it chooses to do the display (e.g. a rendered greyscale image

within an application or displaying the image in a web browser might both be valid for the recipient and faithful to the meaning of the message).

The ordering of the words in an MType SHOULD normally use the object of the message followed by the action to be performed (or the information about that object). For example, the use of "`image.display`" is preferred to "`display.image`" in order to keep the number of top-level words (and thus message classes) like 'image' small, but still allow for a wide variety of messages to be created that can perform many useful actions on an image. If no existing MType exists for the required purpose, developers can agree to the use of a new MType such as "`image.display.extnum`" if, e.g., the ability to display a specific image extension number warrants a new MType.

6.2   The Description of an MType

In order that senders and recipients can agree on what is meant by a given message, the meaning of an MType must be clearly documented. This means that for a given MType the following information must be available:

1. The MType string itself
2. A list of zero or more named parameters
3. A list of zero or more named returned values
4. A description of the meaning of the message

For each of the named parameters, and each of the returned values, the following information must be provided:

- name
- data type (`map`, `list` or `string` as described in Section 3.3) and if appropriate scalar sub-type (see Section 3.4)
- meaning
- whether it is OPTIONAL (considered REQUIRED unless stated otherwise)
- OPTIONAL parameters MAY specify what default will be used if the value is not supplied

Together, this is much the same information as should be given for documentation of a public interface method in a weakly-typed programming language.

The parameters and return values associated with each MType form extensible vocabularies as explained in Section 2.6, except that there is no reserved "`samp.`" namespace.

Note that it is possible for the MType to have no returned values. This is actually quite common if the MType does not represent a request for data. It

is not usually necessary to define a status-like return value (success or failure), since this information can be conveyed as the value of the `samp.status` entry in the call response as described in Section 3.9.

6.3   MType Vocabulary: Extensibility and Process

The set of MTypes forms an extensible vocabulary along the lines of Section 2.6. The relatively small set of MTypes in the "`samp.`" namespace is defined in Section 6.4 of this document, but applications will need to use a wider range of MTypes to exchange useful information. Although clients are formally permitted to define and use any MTypes outside of the reserved "`samp.`" namespace, for effective interoperability there must be public agreement between application authors on this unreserved vocabulary and its semantics.

Since addition of new MTypes is expected to be ongoing, MTypes from this broader vocabulary will be documented outside of this document to avoid the administrative overhead and delay associated with the IVOA Recommendation Track [20]. At time of writing, the procedures for maintaining the list of publicly-agreed MTypes are quite informal. These procedures remain under review, however the current list and details of best practice for adding to it are, and will remain, available in some form from the URL `http://www.ivoa.net/samp/`.

6.4   Core MTypes

This section defines those MTypes currently in the "`samp.`" hierarchy. These are the "administrative"-type MTypes which are core to the SAMP architecture or widely applicable to SAMP applications.

6.4.1   Hub Administrative Messages

The following MTypes are for messages which SHOULD be broadcast by the hub in response to changes in hub state. By subscribing to these messages, clients are able to keep track of the current set of registered applications and of their metadata and subscriptions. In general, non-hub clients SHOULD NOT send these messages.

`samp.hub.event.shutdown:`

> **Arguments:**
>> *none*
>
> **Return Values:**

none

Description:

The hub SHOULD broadcast this message just before it exits. It SHOULD also send it to clients who are registered using a given profile if that profile is about to shut down, even if the hub itself will continue to operate. The hub SHOULD make every effort to broadcast this message even in case of an exit due to an error condition.

`samp.hub.event.register:`

Arguments:

`id` (`string`) — Public ID of newly registered client

Return Values:

none

Description:

The hub SHOULD broadcast this message every time a client successfully registers.

`samp.hub.event.unregister:`

Arguments:

`id` (`string`) — public ID of unregistered client

Return Values:

none

Description:

The hub SHOULD broadcast this message every time a client unregisters.

`samp.hub.event.metadata:`

Arguments:

`id` (`string`) — public ID of client declaring metadata
`metadata` (`map`) — new metadata declared by client

Return Values:

none

Description:

The hub SHOULD broadcast this message every time a client declares its metadata. The `metadata` argument is exactly as passed using the `declareMetadata()` method.

`samp.hub.event.subscriptions:`

Arguments:

> `id` (`string`) — public ID of subscribing client
>
> `subscriptions` (`map`) — new subscriptions declared by client

Return Values:

> *none*

Description:

> The hub SHOULD broadcast this message every time a client declares its subscriptions. The `subscriptions` argument is exactly as passed using the `declareSubscriptions()` method, and hence may contain wildcarded MType strings.

`samp.hub.disconnect:`

Arguments:

> `reason` (`string`) — (OPTIONAL) Short text message indicating the reason that the disconnection is being forced

Return Values:

> *none*

Description:

> The hub SHOULD send this message to a client if the hub intends to disconnect that client forcibly. This indicates that no further communication from that client is welcome, and any such attempts may be expected to fail. The hub may wish to disconnect clients forcibly as a result of some hub timeout policy or for other reasons.

6.4.2   Client Administrative Messages

The following messages are generic messages defined for client use.

`samp.app.ping:`

Arguments:

> *none*

Return Values:

> *none*

Description:

> Diagnostic used to indicate whether an application is currently responding. No "status"-like return value is defined, since in general any response will indicate aliveness, and the normal `samp.status` key in the response may be used to indicate any abnormal state.

`samp.app.status:`

Arguments:

> `txt (string)` — Textual indication of status

Return Values:

> *none*

Description:

> General purpose message to indicate application status.

`samp.app.event.shutdown:`

> **Arguments:**
>
> > *none*
>
> **Return Values:**
>
> > *none*
>
> **Description:**
>
> > Indicates that the sending application is going to shut down. Note that sending this message is not a substitute for unregistering with the hub — registered clients about to shut down SHOULD always explicitly unregister.

`samp.msg.progress:`

> **Arguments:**
>
> > `msgid (string)` — Message ID of a previously received message
> > `txt (string)` — Textual indication of progress
> > `percent (string)` — (OPTIONAL) SAMP float value giving the approximate percentage progress
> > `timeLeft (string)` — (OPTIONAL) SAMP float value giving the estimated time to completion in seconds
>
> **Return Values:**
>
> > *none*
>
> **Description:**
>
> > Reports on progress of a message previously received by the sender of this message. Such progress reports MAY be sent at intervals between the receipt of the message and sending a reply. Note that the `msg-id` of the earlier message must be passed to identify it — the sender of the earlier message (the recipient of this one) will have to have retained it from the return value of the relevant `call*()` method to match progress reports with requests.

A   Changes between PLASTIC and SAMP

In order to facilitate the transition from PLASTIC to SAMP from an application developer's point of view, we summarize in this Appendix the main changes. In some cases the reasons for these are summarized as well.

Language Neutrality: PLASTIC contained some Java-specific ideas and details, in particular an API defined by a Java interface, use of Java RMI-Lite as a transport protocol option, and a lockfile format based on java Property serialization. No features of SAMP are specific to, or defined with reference to, Java (or to any other programming language).

Profiles: The formal notion of a SAMP Profile replaces the choices of transport protocol in PLASTIC.

Nomenclature: Much of the terminology has changed between PLASTIC and SAMP, in some cases to provide better consistency with common usage in messaging systems. There is not in all cases a one-to-one correspondence betweeen PLASTIC and SAMP concepts, but a partial translation table is as follows:

PLASTIC	SAMP
message	MType
support a message	subscribe to an MType
registered application	client
synchronous request	synchronous call/response
asynchronous request	notification

MTypes: In PLASTIC message semantics were defined using opaque URIs such as `ivo://votech.org/hub/event/HubStopping`. SAMP replaces these with a vocabulary of structured MTypes such as `samp.hub.event.shutdown`.

Asynchrony: Responses from messages in PLASTIC were returned synchronously, using blocking methods at both sender and recipient ends. As well as inhibiting flexibility, this risked timeouts for long processing times at the discretion of the underlying transport. The basic model in SAMP relies on asynchronous responses, though a synchronous façade hub method is also provided for convenience of the sender. Client toolkits may also wish to provide client-side synchronous façades based on fully asynchronous messaging.

Registration: In PLASTIC clients registered with a single call which acquired a hub connection and declared callback information, message subscriptions, and some metadata. In SAMP, these four operations have been decomposed into separate calls. As well as being tidier, this offers benefits such as meaning that the subscriptions and metadata can be updated during the lifetime of the connection.

Client Metadata: PLASTIC stored some application metadata (Name) in the hub and provided acess to others (Description, Icon URL, ...) using custom messages. SAMP stores it all in the hub providing better extensibility and consistency as well as improving metadata provision for non-callable applications and somewhat reducing traffic and burden on applications.

Named Parameters: The parameters for PLASTIC messages were identified by sequence (forming a list), while the parameters for SAMP MTypes are identified by name (forming a map). As well as improving documentability, this makes it much more convenient to allow for optional parameters or to introduce new ones. The same arrangement applies to return values.

Recipient Targetting: PLASTIC featured methods for sending messages to all or to an explicit list of recipients. In practice the list variants were rarely used except to send to a single recipient. SAMP has methods for sending to all or to a single recipient.

Typing: Data types in PLASTIC were based partly on Java and partly on XML-RPC types. There was not a one-to-one correspondence between types in the Java-RMI transport and the XML-RPC one, which encouraged confusion. Parameter types included integer, floating point and boolean as well as string, which proved problematic to use correctly from some weakly-typed languages. SAMP uses a more restricted set of types (namely string, list and map) at the protocol level, along with some auxiliary rules for encoding numbers and booleans as strings.

Lockfile: The lockfile in SAMP's standard profile is named `.samp`, its format is defined explicitly rather than with reference to Java documentation, and there is better provision for its location in a language-independent way on MS Windows systems. In many cases however, the same lockfile location/parsing code will work for both SAMP and PLASTIC except for the different filenames (".samp" vs. ".plastic").

Public/Private ID: In PLASTIC a single, public ID was used to label and identify applications during communications directed to the hub or to other applications. This meant that applications could easily, if they wished, impersonate other applications. The practice in SAMP is to use different IDs for public labelling and private identification, which means that such "spoofing" is no longer a danger.

Errors: SAMP has provision to return more structured error information than PLASTIC did.

Extensibility: Although PLASTIC was in some ways extensible, SAMP provides more hooks for future extension, in particular by pervasive use of the *extensible vocabulary* pattern.

B   Change History

Changes to SAMP between Working Draft version 1.0 (2008-06-25) and Recommendation version 1.11 (2009-04-21):

- Return values of `callAll` and `notifyAll` operations changed; they now return information about clients receiving the messages (Section 3.11).
- Characters allowed in `string` type restricted to avoid problems transmitting over XML; was 0x01–0x7f, now 0x09, 0x0a, 0x0d, 0x20–0x7f (Section 3.3).
- New hub administrative message `samp.hub.disconnect` (Section 6.4.1).
- Empty placeholder appendix on SAMP/PLASTIC interoperability removed.
- Wording clarified and made more explicit in a few places.
- Typos fixed, including incorrect BNF in Section 3.7.
- Author list re-ordered.
- Editorial changes and clarifications following RFC period.
- MType Vocabulary section now directs readers to `http://www.ivoa.net/samp/` to find current MType list and process.

Changes to SAMP between Recommendation version 1.11 (2009-04-21) and version 1.2 (2010-12-16):

- Use of new SAMP_HUB environment variable lockfile location option documented in section 4.3.
- Added Non-Technical Preamble section 1.1 as per agreement for all new/revised IVOA documents.

Changes to SAMP between Recommendation version 1.2 (2010-12-16) and version 1.3 (2012-04-11):

- Add a new Section 5 on the Web Profile. Minor changes in the rest of the document noting the existence of this new Profile.
- Add a new Section 2.8 discussing security issues in general, with reference to their particular consideration for both Standard and Web Profiles. The discussion of Standard Profile security is moved to its own new Section 4.3.2.
- MType syntax declaration in Section 3.7 now permits upper-case letters (for consistency with actual usage).
- Sections 3.9 and 3.13 now note that the hub is permitted to generate and forward an error response on behalf of a client under some circumstances. The `samp.code=samp.noresponse` code is reserved for this purpose.

- Section 2.6 now reserves a namespace "`x-samp`" for keys in an extensible vocabulary which are proposed for possible future introduction into this standard.
- A comment has been added to Section 3.3 concerning recommended protocols for use with URLs in messages.

References

[1] C. Arviset et al., "IVOA Architecture", IVOA Note, 2010

[2] F. Bonnarel, P. Fernique, O. Bienaymé, D. Egret, F. Genova, M. Louys, F. Ochsenbein, M. Wenger, and J. G. Bartlett, "The ALADIN interactive sky atlas. A reference tool for identification of astronomical sources", *A&AS*, 143:33–40, 2000

[3] U. Becciani, M. Comparato, A. Costa, C. Gheller, B. Larsson, F. Pasian, and R. Smareglia. "VisIVO: an interoperable visualisation tool for Virtual Observatory data", *Highlights of Astronomy*, 14:622–622, 2007

[4] `http://hea-www.harvard.edu/RD/xpa/`

[5] J. Taylor, T. Boch, M. Comparato, M. Taylor, and N. Winstanley. "PLASTIC — a protocol for desktop application interoperability", IVOA Note, 2006

[6] `http://plastic.sourceforge.net/`

[7] `http://www.xmlrpc.com/`

[8] S. Bradner, RFC 2119: "Key words for use in RFCs to Indicate Requirement Levels", IETF Request For Comments, 1997

[9] T. Berners-Lee, L. Masinter, M. McCahill, RFC 1738: "Uniform Resource Locators (URL)", IETF Request For Comments, 1994

[10] T. Dierks, C. Allen, RFC 2246: "The TLS Protocol", IETF Request For Comments, 1999

[11] A. van Kesteren (Ed.), "Cross-Origin Resource Sharing", W3C Working Draft, 2010

[12] A. van Kesteren (Ed.), "XMLHttpRequest Level 2", W3C Working Draft, 2012

[13] Adobe Flash cross-domain policy, `http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html`

[14] Microsoft Silverlight cross-domain policy, `http://msdn.microsoft.com/en-us/library/cc645032(VS.95).aspx`

[15] T. Berners-Lee et al., RFC 2396: "Uniform Resource Identifiers (URI): Generic Syntax", IETF Request For Comments, 1998

[16] R. Fielding et al., RFC 2616: "Hypertext Transfer Protocol – HTTP/1.1", IETF Request For Comments, 1999

[17] D. Eastlake et al., RFC 3275: "XML-Signature Syntax and Processing", IETF Request For Comments, 2002

[18] A. Barth, "The Web Origin Concept", IETF Draft, 2010

[19] A. Preite Martinez et al. "An IVOA Standard for Unified Content Descriptors", IVOA Recommendation, 2007

[20] R. J. Hanisch et al. "IVOA Document Standards", IVOA Recommendation, 2010